KIWIGRID
CONNECTING ENERGY

HOCHSCHULE
MITTWEIDA
UNIVERSITY OF
APPLIED SCIENCES

Bachelorarbeit

“Evaluierung von BigData Warehouse Technologien für die Speicherung und Berechnung von Zeitserien über Sensordaten”

Fakultät Elektro- und Informationstechnik

Abgegeben von: Julian Wissmann
Matr. Nr. 28854

Abgegeben am:

Betreuer: Prof. Dr. Ing Wilfried Schmalwasser, Dipl. Inf. Jörg Eichhorn

Inhaltsverzeichnis

Illustrationsverzeichnis

Tabellenverzeichnis

1. Einleitung

1.1 Zielsetzung

Ziel dieser Arbeit ist es, ein geeignetes Datenbanksystem zum Einsatz als Data-Warehouse für Zeitseriendaten bei der Kiwigrid GmbH zu finden. Dazu werden geeignete Datenbanksysteme und ein existierendes Zeitserien-Datenbank-System betrachtet um deren Eignung zu untersuchen. Dabei soll eine Technologie ermittelt werden, auf deren Basis das aktuelle, MySQL basierte Data-Warehouse abgelöst werden kann.
Die dazu notwendigen Untersuchungen sollen anhand realer Daten aus dem Kiwigrid Data-Warehouse stattfinden. Das aufgebaute Testsetup soll möglichst ähnlich aufgebaut sein wie ein Setup, welches man produktiv einsetzen würde.
Das bedeutet, dass das Setup eine mindestens dreifache Redundanz der Daten miteinbezieht und auch gegen Ausfälle einzelner Knoten abgesichert ist.
Am Ende soll so eine Empfehlung erarbeitet werden, welches System sich am besten für weitere Entwicklungsbemühungen im Bereich Data-Warehousing eignet.

Wichtig sind hierbei Datenzugriffsmöglichkeiten, die es erlauben, Daten möglichst flexibel und schnell zu aggregieren. Der benötigte Speicherplatz für die Persistierung der Zeitseriendaten sowie die Bearbeitungsgeschwindigkeit der Abfragen sind weitere wichtige Faktoren. Zudem soll die Skalierbarkeit des Systems hin zur Bearbeitung tausender paralleler Anfragen auf einen permanent wachsenden Datenbestand abgeschätzt werden.

1.2 Gliederung der Arbeit

Diese Arbeit ist in 6 Abschnitte unterteilt. Der erste Abschnitt dient zur Einleitung und zur Beschreibung der Zielstellung.

Im zweiten Abschnitt werden technische Grundlagen und Begriffe erläutert. Darüber hinaus werden die zu evaluierenden Systeme und ihre zugrunde liegenden Technologien und Konzepte vorgestellt. Es wird weiterhin darauf eingegangen, wie ein Setup dieser Systeme aussehen sollte und abgeschätzt, wie groß der damit verbundene administrative Aufwand ist.

Der dritte Abschnitt widmet sich der Analyse. Dort werden aus einer Analyse des Ist-Zustandes Ziele der Evaluation und Anforderungen an die Systeme erarbeitet. Außerdem werden hier die Testkriterien für die zu evaluierenden Systeme ermittelt. Basierend auf den erarbeiteten Kriterien werden dann konkrete Testfälle definiert, die zum Vergleich der Systeme in der Testdurchführung herangezogen werden.

Im vierten Abschnitt wird die Testdurchführung behandelt. Hier wird auf die verwendete Methodik, die Konfigurationen und die Testdurchführung eingegangen.
Im fünften Abschnitt werden die Ergebnisse der durchgeführten Tests verglichen und ausgewertet.

Der letzte Abschnitt beinhaltet eine abschließende Betrachtung und Zusammenfassung der Arbeit. Es wird eine Empfehlung auf Basis der durchgeführten Evaluation gegeben. Außerdem wird festgestellt ob alle am Anfang gestellten Fragen beantwortet und die festgelegten Ziele erreicht wurden. Abschließend erfolgt ein Ausblick auf mögliche zukünftige Entwicklungen des Projekts.

2. Grundlagen

2.1 NoSQL Datenbanken

Während SQL basierte Systeme gut geeignet sind, um strukturierte Daten entsprechend dem relationalen Modell, welches Edgar F. Codd in seinem Fachartikel “A Relational Model of Data for Large Shared Data Banks” [Codd, E.F. 1970] beschreibt, zu verwalten, bedarf es heute häufig einer besseren Lösung für den effizienten Umgang mit unstrukturierten oder schemalosen Daten. Um solche schemalosen Daten handelt es sich zum Beispiel bei Zeitseriendaten und häufig auch bei Objekten, die aus Anwendungen heraus persistiert werden. In diesem Bereich haben sich zunehmend sogenannte NoSQL-Datenbanken etabliert, welche sich dadurch auszeichnen, andere Zugriffsmechanismen als tabellarische Relationen zu verwenden. Auch werden häufig andere Persistierungsmechanismen als der in relationalen Datenbanken verbreitete B-Baum verwendet. Viele NoSQL Systeme setzen hier auf LSM-Bäume.
NoSQL Datenbanken kann man in die Kategorien Key-Value Stores, Graphdatenbanken und dokumentenorientierte Datenbanken unterteilen. Meist werden bei diesen Systemen Ziele wie die Vereinfachung des Schemas oder verbesserte horizontale Skalierbarkeit verfolgt. Häufig werden bei NoSQL-Systemen auch Kompromisse im Bereich Konsistenz eingegangen um eine bessere Verfügbarkeit und Partitionstoleranz zu erreichen [vgl. Gilbert, S., Lynch, N. 2002, 5. Conclusion]. Dadurch konnten sich diese Systeme in den Bereichen Big-Data und in großen Webanwendungen etablieren, wo die Anforderungen an Skalierung und Verfügbarkeit die Konsistenzanforderungen in der Regel übersteigen. [vgl. George, L. 2011. S. 8-13]

2.1.1 Key-Value Stores

Diese Systeme setzen ein sehr einfaches Zugriffsmodell um, welches es erlaubt, Paare von Werten (Value) und Zugrifsschlüsseln (Key) abzulegen. Der Zugriff auf einen Wert erfolgt dabei anhand des Schlüssels. Eine weiterentwicklung dieses Schemas stellt das von Fay Chang et. al. Vorgestellte BigTable System dar, welches eine Zordnung von einem Sclüssel zu belieb vielen Werten erlaubt, wodurch effektiv gesehen eine spaltenorientierte Datenbank entsteht. Der Zugriff auf Einzelwerte wird dabei durch eine

zusätzliche Eigenschaft, den sogenannte Column Qualifier sichergestellt. Um ein solches System handelt es sich bei dem in 2.4 vorgestellten HBase. [vgl. Redmond, E.,Wilson, J., 2012, S. 4]

2.1.2 Dokumentenorientierte Datenbanken

Bei Dokumentenorientierte Datenbanken sind Dokumente die kleinste Speichereinheit. Entgegen relationalen Datenbanken, deren Tabellen einem festen Schema unterliegen, sind Dokumente Schemafrei. Das bedeutet, dass diese zwar aus Applikationssicht einem festen, strukturierten Dateiformat unterliegen können, dies jedoch aus Sicht der Datenbank nicht müssen. Die Struktur obliegt alleine dem verwendeten Datei- oder Datenformat.
Das in 2.6 vorgestellte MongoDB ist ein solches System. [vgl. Redmond, E.,Wilson, J., 2012, S. 5-6]

2.2 Verteilte Systeme

Verteilte Systeme bilden die Grundlage vieler NoSQL-Systeme, neben vielen weiteren können HBase und MongoDB als solche klassifiziert werden.
Nach Tanenbaum und van Steen kann ein verteiltes System wie folgt definiert werden:

„Ein verteiltes System ist eine Sammlung von unabhängigen Computern die für den Nutzer als ein einziges, Kohärentes System erscheinen.“
(Übersetzung des Autors) [Tanenbaum, A., van Steen, M., 2007, S. 2]

Daraus geht hervor, dass ein verteiltes System aus einer Ansammlung von miteinander verbundenen Systemen besteht. Diese stellen dem Anwender gemeinsam eine Anwendung zur Verfügung und vermitteln diesem das Gefühl, dass er lediglich mit einem einzelnen System interagiert. Ziel ist es, den Ressourcenzugriff für den Nutzer zu vereinfachen und die Organisation und Komunikation innerhalb des Systems vor ihm zu verbergen. [vgl. Tanenbaum, A., van Steen, M., 2007, S. 2]

Die Eigenschaft eines verteilten Systems, sich gegenüber dem Nutzer oder einer Anwendung als einzelnes System auszugeben, nennt man Transparenz. Diese

Eigenschaft kann auf verschiedene Aspekte verteilter Systeme angewendet werden. Eine Auflistung der wichtigsten Aspekte ist Tabelle 1 zu entnehmen:

Transparenz	**Beschreibung**
Zugriff	Verbirgt Unterschiede in der Darstellung und die Art und Weise, wie auf eine Ressource zugegriffen wird
Ort	Verbirgt, wo sich eine Ressource befindet
Migration	Verbirgt, dass eine Ressource an einen anderen Ort verschoben werden kann
Relokation	Verbirgt, dass eine Ressource an einen anderen Ort verschoben werden kann, während sie genutzt wird
Replikation	Verbirgt, dass eine Ressource repliziert ist
Nebenläufigkeit	Verbirgt, dass eine Ressource von mehreren konkurrierenden Benutzern gleichzeitig genutzt werden kann
Fehler	Verbirgt den Ausfall und die Wiederherstellung einer Ressource

Tabelle 1: Unterschiedliche Formen der Transparenz (Übersetzung des Autors) [Tanenbaum, A., van Steen, M., 2007, S. 5]

Anwendung finden verteilte Systeme häufig im Bereich des Hochleistungsrechnens (z. B. Seti@Home), im Internet (z. B. DNS) aber auch in klassischen Büroumgebungen wo datenbankgestützte Anwendungen eine wichtige Rolle spielen.

2.3 Zeitserien

Die Sammlung von Zeitseriendaten ist wichtiger Bestandteil vieler wissenschaftlicher, wie auch ökonomischer Prozesse. Viele Erkenntnisse lassen sich sich aus solchen Daten gewinnen. Zeitserien können wie folgt definiert werden:

“Eine geordnete Sequenz von Werten einer Variable über ein konstantes Intervall.” (Übersetzung des Autors)[NIST, 2013, Kapitel 6.4.1]

Als Beispiele für Zeitseriendaten lassen sich periodische Pegelstandsmessungen an Flüssen, die Messung der täglichen Maximal- und Minimaltemperatur oder minutenaktuelle Stromerzeugungswerte, wie sie, neben einer Vielzahl anderer Daten, von den

Energiemanagern der Kiwigrid GmbH erfasst werden, heranziehen. Gemeinsam haben diese Daten die feste Periodizität und die daraus folgende gute Analysierbarkeit. So ist es mit geeigneten mathematischen Verfahren möglich anhand von Zeitseriendaten Trends zu erkennen, die zyklische Wiederkehr von Ereignissen zu bestimmen und somit letztlich ein besseres Verständnis eines Systems zu gewinnen. [vgl. Universität Würzburg, 2012, S. 1]

2.4 Apache HBase

Apache HBase ist eine verteilte, spaltenorientierte Datenbank, die auf Apache Hadoop und Apache ZooKeeper aufsetzt. Sie ermöglicht das Lesen und Schreiben auf sehr große Datensätze in Echtzeit.
Apache HBase kann als Nachbau von Google BigTable angesehen werden. Es orintiert sich sehr stark an dem von Chang et al. vorgestellten Fachartikel „BigTable: a distributed storage system for structured data". [Chang, F., et al. 2008] Es implementiert das dort vorgestellte Datenhaltungsschema und viele weitere Designentscheidungen, die Google in BigTable traf, wodurch es sehr gut für den Einsatz in Umgebungen geeignet ist, die horizontale Skalierbarkeit, Hochverfügbarkeit und Konsistenz voraussetzen . [vgl. Apache HBase. 2015. Home]

In Bezug auf das CAP-Theorem ist HBase ein CP-System. Es verzichtet auf Verfügbarkeit um Konsistenz und Partitionstoleranz bieten zu können.

2.4.1 Apache Hadoop

Hadoop ist ein Framework bestehend aus verschiedenen Softwarekomponenten, auf dessen Basis sich skalierbare, verteilt arbeitende Software entwickeln lässt. Im wesentlichen handelt es sich um einen Nachbau der 2003 von Google vorgestellten Google File System [Ghemawat, S., Gobioff, H., Leung, S. 2003] und des ebenfalls von Google 2004 vorgestellten MapReduce [Dean, J., Ghemawat, S. 2004].

HDFS

Als grundlegende Komponente von Hadoop kann HDFS, das Hadoop Distributed Filesystem gesehen werden. Es ist ein sehr speziell auf die Funktionsweise weiterer

Hadoop Komponenten zugeschnittenes, verteiltes Dateisystem. So ist es stark auf Datenströme optimiert und bietet in Unterstützung mit Diensten, die auf dem Dateisystem aufsetzen, Datenlokalität. Das bedeutet, dass in der Datenverarbeitung mit Hadoop versucht wird, Algorithmen immer dort auszuführen, wo sich auch die Daten befinden. Daraus resultieren ein relativ geringer Netzwerkoverhead und eine insgesamt schnellere Ausführung, da im allgemeinen nicht auf Datenübertragungen gewartet werden muss.
Ein weiterer Fokus des Dateisystems liegt auf der Datensicherheit. Diese wird mittels Replikation im Cluster erzeugt. Daten gelten erst als Sicher, sobald mindestens 3 Replikas existieren. [vgl. White, T. 2011, S. 41f]

YARN

YARN (Yet Another Resource Negotiator) ist ein noch relativ junges Projekt. Ursprünglich als Teil von MapReduce entwickelt hat sich dieses Projekt mittlerweile zu einem allgemeinen Ressourcenmanager für verteilte Anwendungen entwickelt. Mit Hilfe von YARN ist es möglich, bereitgestellte Ressourcen auf verschiedene Anwendungen oder aber mehrere Instanzen einer Anwendung zu verteilen. [vgl. Apache Hadoop. 2013. Apache Hadoop NextGen MapReduce]

MapReduce

Dabei handelt es sich um ein ursprünglich von Google vorgestelltes Programmiermodell zur Ausführung nebenläufiger Berechnungen mit Hilfe von Clustern [White, T. 2011, S. 15]. Aufgrund seiner Fähigkeit große Datensätze effizient aufzuteilen ist es besonders für den Umgang mit großen Datensätzen (hunderte von GB bis hin zu mehreren PB) geeignet.

2.4.2 Funktion von HBase

HBase ist eine in Java geschriebene Anwendung. Der Server besteht, als verteiltes System aus den drei Grundkomponenten Master, RegionServer und Client. Während dabei der Master hochverfügbar betrieben werden kann verlassen sich die an einem Cluster beteiligten RegionServer zur Sicherstellung von Datensicherheit und zur Wiederherstellung von Daten im Falle des Ausfalls eines RegionServers auf das darunter liegende HDFS.

HBase Master

Der HBase Master ist im Cluster für Koordination und Administration zuständig. Das bedeutet, dass der Master für die Erstellung von Tabellen, und die Zuweisung von Regions an die RegionServer zuständig ist.
Während es möglich ist, mehrere Master in einem Cluster zu betreiben, kann zu jedem gegebenen Zeitpunkt lediglich ein Master aktiv sein. Welcher dies ist, wird mit Hilfe der von ZooKeeper bereitgestellten Leader-Election bestimmt.. [vgl. Apache HBase Dokumentation, Kapitel 63. Master]

HBase RegionServer

Der RegionServer ist für die eigentliche Datenhaltung zuständig. Er verwaltet die sogenannten Regions. Regions sind ein Zentrales Element für die Verfügbarkeit und Verteilung von Tabellendaten. Sie verwalten sogenannte Stores, Speichereinheiten bestehend aus Storefiles, die in HDFS abgeelgt sind und MemStores zum Cachen von Daten. Regions verwalten dabei einen sortierten Bereich von Reihen aus einer Tabelle, in seltenen Fällen, wenn eine Tabelle sehr klein ist möglicherweise auch eine komplette Tabelle. Erreicht eine Region eine definierte Größengrenze, so veranlasst sie einen Region Split. Dieser hat ihre Teilung in zwei Regions ungefähr gleicher Größe zur Folge. [vgl. Apache HBase Dokumentation, Kapitel 65.4 Region Splits]

RegionServer behandeln alle Lese- und Schreiboperationen auf alle ihnen zugewiesenen Regions. In einem Cluster kann es eine nahezu beliebige Anzahl an RegionServern geben, die selbst wiederum bis zu mehreren hundert Regions verwalten können. [vgl. Apache HBase Dokumentation, Kapitel 64. RegionServer]

Durch dieses Design ist HBase dazu in der Lage, über tausende Knoten horizontal zu skalieren. Das bedeutet, dass eine verbesserte Skalierung, durch das hinzufügen weiterer Knoten basierend auf durchschnittlicher Hardware erreicht werden kann. Dies steht im Gegensatz zur vertikalen Skalierung, die im bereich der relationalen Datenbanken verbreitet ist, wo eine weitere Skalierung dadurch erreicht wird, dass aktuell genutzte Server durch leistungsfähigere Modelle ausgetauscht werden.

Coprocessors

Bei dem sogenannten „Coprocessor Framework“ handelt es sich um eine Sammlung generischer Schnittstellen auf HBase Master- und Regionebene. Diese können genutzt werden, um eigene Funktionen zu implementieren, die Serverseitig ausgeführt werden sollen [vgl. Lai, M., Koontz, E., Purtell, A. 2012].
Die Ausführung eines solchen Coprocessors erfolgt dabei parallel auf allen Regionen des Clusters.
Besonders interessant ist dies für weniger komplexe Berechnungen wie die Bestimmung von Durchschnitts- oder Summenwerten über zehntausende Einträge. Durch die Ermittlung solcher Ergebnisse durch die Regions kann die Last der Clients, wie auch der notwendige Netzwerkverkehr stark reduziert werden. Für die Region Server bedeutet dies sogar eine bessere Auslastung vorhandener Ressourcen, da deren Hauptlast I/O gebunden und nicht CPU gebunden ist.

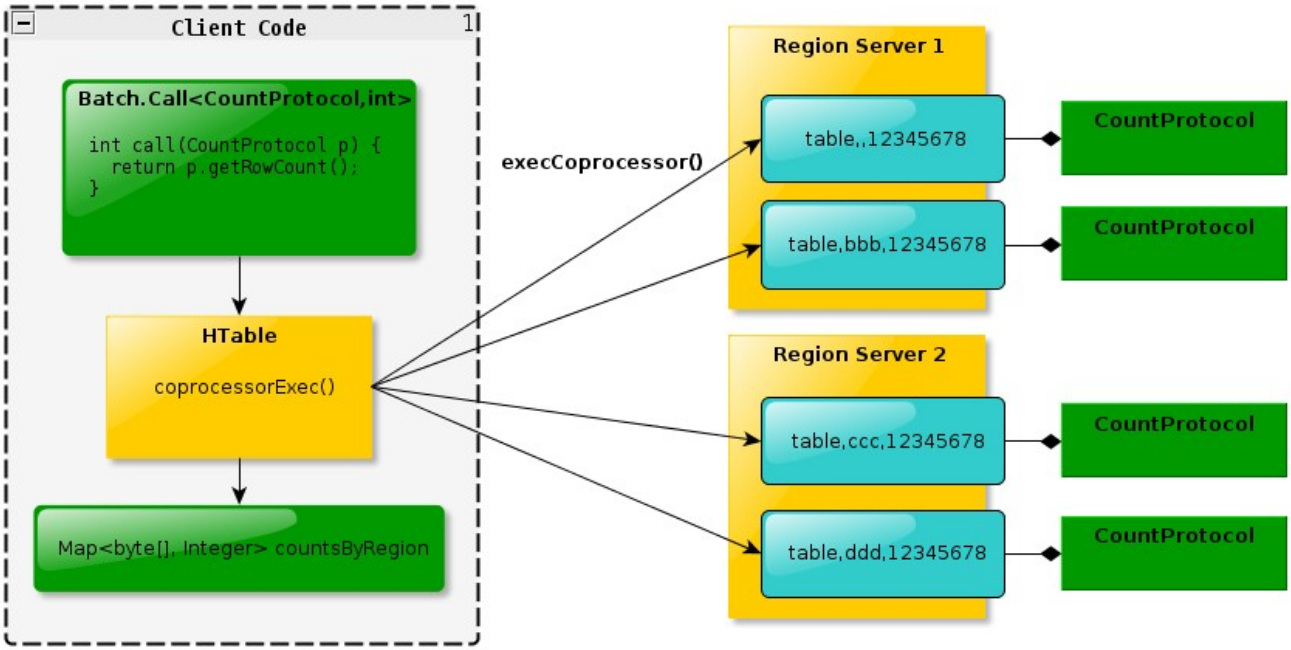

Illustration 1: Coprocessor Ausführung [Lai, M., Koontz, E., Purtell, A. 2012]

Datenspeicherung

Im Gegensatz zu den meisten relationalen Datenbanksystemen setzt HBase nicht auf B oder B+ Bäume als grundlegende Datenstruktur, sondern auf LSM-Bäume. [vgl. George,

L. 2011. S. 24]

2.5 OpenTSDB

„OpenTSDB ist eine verteilte, skalierbare Zeitseriendatenbank auf Basis von HBase. OpenTSDB wurde geschrieben, um um eine allgemeine Notwendigkeit zu addressieren: Speicherung, Indexierung und Bereitstellung von Metriken die von Computersystemen (Netzwerkequipment, Betriebssysteme, Applikationen) in großem Maßstab erzeugt werden, sowie die Vereinfachung des Zugriffs und der grafischen Aufbereitung dieser Daten.“ (Übersetzung des Autors) [Dimiduk, Khurana S. 182]

OpenTSDB entstand 2010 bei StumbleUpon als Werkzeug zum Infrastrukturmonitoring. Ziel der Entwicklung war es, Soft- und Hardwaremetriken aus StumbleUpons Serverfarm verarbeiten- und speichern zu können. StumbleUpon gibt an, mehr als eine Milliarde Datenpunkte pro Tag auf einem Cluster mit 5 Knoten mittels OpenTSDB zu erfassen. Eine Besonderheit von OpenTSDB ist die Verwendung von Asynchbase, einem im Rahmen der OpenTSDB-Entwicklung entstandenen asnychronen HBase-Client. Die OpenTSDB-Entwickler behaupten, dass dieser mehr als doppelt so schnell ist, wie der normale, HTable genannte, HBase Client [vgl. Sigoure, B. 2012. Slide 6]. Asynchbase unterstützt jedoch nicht den vollen Funktionsumfang von HBase, davon sind insbesondere neuere Funktionen wie die Coprocessor-Ausführung betroffen.

2.5.1 Funktion von OpenTSDB

Die zentrale Komponente von OpenTSDB ist der TSD, der HBase zur Datenspeicherung verwendet. Es können beliebig viele, voneinader unabhängige Instanzen des TSD betrieben werden. Die Empfohlene Herangehensweise ist dabei, Schreiben und Lesen auf eigene TSDs aufzuteilen. Der TSD bietet neben einer Telnetartigen API auch eine HTTP-REST Schnittstelle, sowie eine einfache Weboberfläche zur Visualisierung von Daten.
Aggregationen führt der TSD mangels Unterstützung für Coprocessors Clientseitig aus. Unterstützt werden Downsampling, Interpolation und Aggregation.

2.6 MongoDB

MongoDB ist eine dokumentenorientierte Datenbank. Das verwendete Datenformat ist BSON (Binary JSON). MongoDB ist schemalos, jedoch wird die Indexierung von Feldern unterstützt. Der Datenbankzugriff erfolgt in einer einfachen, JavaScript basierten Zugriffssyntax. Um auch komplexere Funktionen abbilden zu können ist in MongoDB ein JavaScript Interpreter integriert. Dadurch wird auch ermöglicht, Scripte in der Datenbank zu hinterlegen und auszuführen. [vgl. Redmond, E., Wilson R. 2012. S. 135, 137]

Bezogen auf das CAP-Theorem kann MongoDB als CP-System klassifiziert werden, jedoch gibt es bei der Konsistenz einige Kompromisse, die zu beachten sind [vgl. MongoDB Blog, On Distributed Consistency]

2.6.1 Funktionsweise von MongoDB

MongoDB kann, muss jedoch nicht als verteilte Datenbank betrieben werden. Im verteilten Betrieb (Sharding) setzt MongoDB ähnlich wie BigTable auf einen Fragmentierungsschlüssel anhand welchem die Daten aufgeteilt werden. Auf ein verteiltes Dateisystem wird jedoch verzichtet. Zum Erreichen von Datensicherheit wird daher auf die mehrfache Replikation mittels Standby-Datenbankservern (Replication) gesetzt. In einem verteilten Setup kommen so bis zu vier verschiedene Nodetypen vor. [vgl. Redmond, E., Wilson R. 2012. S. 231ff]

MongoDB Server (mongod)

Der mongod ist die Hauptkomponente einer jeden MongoDB Installation. In einem Einzelknoten-Setup stellt er alleine alle notwendigen Funktionen zur Verfügung. Um einen mongod zu replizieren wird eine ungerade Anzahl von mongod Knoten verwendet. Diese können selbständig untereinander ein Quorum bestimmen. In einem verteilten Setup bildet eine solche Replication Group einen Bereich (Shard) ab. Das heißt eine verteilte Datenbank besteht immer aus mehreren Replication Groups

MongoDB Arbiter

Beim Arbiter handelt es sich um einen speziell konfigurierten mongod. Er speichert selbst keine Daten, identifiziert sich einem Replication Set gegnüber jedoch als Mitglied. Dadurch wird es möglich Replication Sets auch mit weniger als 3 Knoten zu betreiben, bzw. in einem Setup mit einer geraden Knoten-Anzahl die Quorum-Bildung zu ermöglichen. [vgl. MongoDB Documentation, Replica Set Arbiter]

Config Server

Der Config Server ist eine spezielle Rolle des mongod. Config Server verwalten ausschließlich Sharding Metadaten. Das heißt sie speichern, in welcher Replication Group (Shard) sich welche Daten befinden. Eine weitere Besonderheit ist, dass in einem verteilten Setup immer exakt drei Config Server vorkommen müssen. [vgl. Redmond, E., Wilson R. 2012. S. 242]

Mongos Router

Wie der Config Server wird auch der mongos ausschließlich in einem verteilten Setup benötigt. Er dient dann als Frontend für den Datenbankzugriff. Ihm sind alle Shards, sowie die Config Server bekannt, so kann er Anfragen auf die Shards aufteilen und beim Schreiben die Metadaten der Config Server anpassen. [vgl. MongoDB Documentation, mongos]

2.6.2 Datenspeicherung

Bis einschließlich Version 2.6 nutzte MongoDB ausschließlich B-Bäume für die Indizierung von Daten. Ab Version 2.8 werden mit der neu eingeführten WiredTiger Datenbank-Engine jedoch auch LSM-Bäume unterstützt. Damit ist eine deutliche Verbesserung der Schreibleistung von MongoDB zu erwarten. Darüber hinaus unterstützt WiredTiger Snappy-Kompression, wodurch sowohl eine Reduzierung des benötigten Festplattenplatzes, sowie auch eine Durchsatzsteigerung aufgrund der reduzierten Datengröße zu erwarten ist. Außerdem wird nun Locking auf Dokumentenebene unterstützt, dadurch wird bei Inserts nicht mehr die gesamte Collection gesperrt. [vgl. Wolpe, T. 2014]

In einem verteilten Setup gibt es in MongoDB einen primären Shard, auf den alle neuen Daten, die nicht bereits einem anderen Shard zuschreibbar sind, geschrieben werden. In regelmäßigen Abständen werden diese dann durch einen Balancer genannten Prozess im Cluster verteilt. Dieser ist Teil des Mongos Routers und teilt sich das Write Lock mit den ebenfalls laufenden Inserts. [vgl. MongoDB Documentation, Sharded Collection Balancing]

2.8 Operativer Aufwand

Den tatsächlichen operativen Aufwand der Systeme auf Basis der vorgenommenen Evaluierung abzuschätzen ist schwierig. Es ist hauptsächlich möglich, auf den Setupprozess einzugehen.

Setup HBase

Das Setup von Apache HBase, bzw. generell des kompletten Hadoop Stacks ist aufgrund seiner komplexen Abhängigkeitsstruktur nicht trivial. Um korrekt zu funktionieren werden DNS, Zookeeper, sowie HDFS und YARN aus dem Hadoop Projekt benötigt.

Ein HBase Setup besteht aus mindestens 5 Knoten. Auf einer ungeraden Anzahl aber mindestens 3 davon läuft der Koordinierungsdienst ZooKeeper. Zwei Knoten sind Hadoop NameNode, sowie HBase Master. Auf allen befindet sich ein HDFS DataNode und ein HBase RegionServer. Weitere Knoten können nach Bedarf hinzugefügt werden.

Es ist bei den Hadoop Komponenten wichtig, dass die verwendeten Softwareversionen zueinander passen. Zudem müssen Änderungen an Konfigurationsparametern in vielen Fällen auf alle Knoten angewendet werden, was bei einer händischen Administration der einzelnen Clusterknoten sehr aufwändig und Fehleranfällig ist.

Ein weiterer wichtiger Faktor für das Setup und den erfolgreichen Betrieb ist ein gutes Monitoring, da viele Probleme wie Split Brain sich unter Umständen nicht direkt bemerkbar machen.

Da diese Probleme allgemein bekannt sind, bieten die Hersteller Hadoop basierter Softwarelösungen häufig eigene Lösungen für das Deployment und Monitoring des

Hadoop Stacks an. Eine solche Lösung ist Hortonworks Ambari, welches mittlerweile als Open Source Projekt an die Apache Software Foundation übergeben wurde. Dieses wird für das Deployment und Monitoring von Hadoop und HBase verwendet.

Setup MongoDB

Da MongoDB keine komplexen Abhängigkeiten für den Betrieb hat, ist ein Setup entsprechend einfach. Um die Voraussetzungen des Sharding zu erfüllen ist jedoch eine gute Planung vonnöten.

Ein verteiltes MongoDB Setup besteht aus mindestens 9 Knoten. Davon sind immer genau 3 als ConfigServer vorgesehen, aus den restlichen werden Shars gebildet. Ein solcher Shard besteht aus einer Replication Group, welche aus einer ungeraden Anzahl, aber mindestens 3 Servern besteht. Zusätzlich wird mindestens 1 Mongos-Router benötigt, der mit Hilfe der ConfigServer das Sharding orchestriert.

Diese Komplexität ist darin begründet, dass Sharding und damit horizontale Skalierung kein unmittelbares Designziel von MongoDB war, sondern in Version 1.6 nachgereicht wurde.
Die Konfiguration von MongoDB wird in der Datenbank selbst vorgenommen, für Administration und Automatisierung ist also immer auch ein direkter Zugang vonnöten.
Auch bei MongoDB ist ein gutes und umfangreiches Monitoring unabdingbar. Insbesondere das Sharding benötigt dabei eine umfangreiche Überwachung um Probleme insbesondere im Region Balancing frühzeitig erkennen zu können.

Allgemein

Es lässt sich also sagen, dass trotz der architektonischen Unterschiede bei HBase und MongoDB die operative Herangehensweise unter der Voraussetzung dass beide Plattformen lediglich als Datenbank betrachtet werden, ähnlich ist. Für beide Systeme ist ein umfassendes Monitoring von Festplattendurchsatz, Dateisystem, Netzwerk und verschiedenen Parametern der Software sehr wichtig um mögliche Fehler erfassen zu können. Für beide Systeme gibt es dazu eine Reihe von Plugins die in bestehende Monitoringsysteme integriert werden können.

Die größte Umstellung und Herausforderung aus operativer Perspektive ist die Sicherstellung der dauerhaften Sicherheit und Verfügbarkeit der Systeme. Beide sind für die Verarbeitung großer Datenmengen in verteilten Setups ausgelegt. Dabei setzen sie auf das Verständnis, dass ab einer bestimmten Datenmenge Backups nicht mehr praktikabel sind und der dauerhafte Betrieb des Systems ohne Ausfälle gewährleistet können werden muss. Daher verfügen sie über eine vielzahl von Automatismen, die den Betrieb möglichst fehlertolerant und wartungsarm gestalten sollen.
Auch bei Upgrades wird dies ersichtlich, da beide Systeme einen Rolling Upgrade Prozess unterstützen, bei dem der Cluster Server für Server ohne Downtime auf die neue Version aktualisiert wird.

Eine weitere große Herausforderung stellt die Performanceoptimierung dar. Hier existieren bei beiden eine Vielzahl von Parametern mittels derer eine Optimierung möglich ist. Um diese sinnvoll nutzen zu können ist jedoch ein tiefes Verständnis des Systems und der vorhandenen Daten notwendig.

Im Falle von HBase liegt jedoch der Schluss nahe, das System nicht lediglich als Datenbank zu betrachten. Seine Abhängigkeit von Hadoop bietet vielfältige Möglichkeiten zur Datenverarbeitung neben HBase. So ermöglicht es Stream-Verarbeitung und Batch-Verarbeitung. HDFS kann darüber hinaus als Storage für vielfältige Anwendungen und Aufgaben verwendet werden und nicht zuletzt durch die Entwicklung von YARN erhebt es immer mehr den Anspruch, eine ganzheitliche Cloud-Lösung darzustellen. So ist es seit Kurzem auch möglich, beliebige Applikationen und gar virtuelle Maschinen via YARN zu verwalten [vgl. Seethana, S. 2014]. Betrachtet man all diese Aspekte und zielt darauf, diese auch zu nutzen, steigt damit natürlich auch der operative Aufwand.

3. Analyse

In diesem Abschnitt sollen aus dem Ist-Zustand des aktuellen Warehouse-Systems Anforderungen abgeleitet werden, die dann in Schemas für die zu testenden Systeme überführt werden.
Ziel der Analyse ist es, ein System zu finden, das dazu geeignet ist, das aktuelle, SQL-basierte Warehouse featureseitig zu ersetzen und gleichzeitig mit größer werdendem Bedarf effizient horizontal zu skalieren.

3.1 Anforderungsanalyse

Die Anforderungen für die Systeme leiten sich weitgehend aus vorhandenen Features des aktuellen, SQL basierten Warehouse, wie aus Featurewünschen in der Weiterentwicklung der Kiwigrid Plattform ab. Dabei soll es mit den zu testenden Systemen möglich sein, zunächst ein vergleichbares Feature Set abzubilden, perspektivisch soll jedoch auch ermittelt werden, wie sich dieses erweitern lässt.
Der Fokus der Anforderungsanalyse liegt darauf, zu ermitteln, ob das erstellte Schema alle gewünschten Features theoretisch unterstützten kann.

3.1.1 Ausgangszustand

Bisher kommt zur Speicherung aller Geschäfts- und Zeitseriendaten MySQL zum Einsatz.
Dabei werden Zeitseriendaten in Form von sogenannten Datapoints von Endgeräten an eine Webplattform (Portal) gesendet. Ein Datapoint besteht aus einem Zeitstempel, einem Wert und Metadaten zur eindeutigen Identifikation des zugehörigen Geräts und Datentyps der Zeitreihe (z.B. kWh). Im Portal findet eine Plausibilisierung statt, bevor ein Datenpunkt in einer „realtime_facts“ genannten Tabelle des Warehouse abgelegt wird. Dabei wird eine Verknüpfung zu einem in der Geschäftslogik hinterlegten Datapoint hergestellt, sodass in den realtime_facts lediglich eine ID (Autoincrement), der Zeitstempel, die Datapoint ID, sowie der Wert hinterlegt werden müssen.
Die dort abgelegten Datenpunkte werden dann zu „minute_facts“, „hour_facts“, „day_facts“, sowie „month_facts“ aggregiert. Aus historischen Gründen ist ein Datapoint jedoch wenig aussagekräftig, da er eins zu eins einen von einem Gerät gelesenen Wert

repräsentiert. So ist es möglich, dass die Power Datenpunkte zweier Wechselrichter nicht unmittelbar vergleichbar sind, wenn sie in einem Fall in Watt, im anderen in Kilowatt geführt werden, oder wenn ein solcher Wert für ein Gerät erst aus mehreren realtime_facts berechnet werden muss.
Daher findet bei dieser Aggregation eine nochmalige Transformation statt, sodass am Ende vergleichbare Daten zur Verfügung stehen. Diese transformierten Datenpunkte werden Tags genannt und sind über eine Klasse von Geräten, z.B. Wechselrichter, vergleichbar.
Da die Logik zur Bildung von Tags bereits seit geraumer Zeit direkt auf den Energiemanagern verfügbar ist und dort die Verwendung von Datenpunkten bereits API-seitig verdrängt hat, sind Datenpunkte semantisch für die Entwicklung des Zukünftigen Warehouse irrelevant. Sie dienen jedoch als Datengrundlage für die in dieser Arbeit durchgeführten Tests.

Durch eine grundlegende Neuentwicklung des Portals wird derzeit bereits der Grundstein dafür gelegt, zukünftig komplett auf Datapoints verzichten zu können. Dabei müssen jedoch auch wieder Tags in Form von realtime_facts gespeichert werden. Es fällt lediglich die aufwändige Datapointtransformation weg.
Die Probleme der vielschichtigen Speicherung voraggregierter Werte und die daraus resultierenden starren Aggregationsintervallen bleiben bestehen.
So belegen die, in dem hier verwendeten Datenbankauszug vorhandenen 191.929.820 realtime_facts in MySQL 9,9GB Platz, hinzu kommen 6,2GB Indizes, die für einen schnellen Zugriff notwendig sind. Bei den minute_facts hingegen ergibt sich bereits ein anderes Bild. Hier kommen auf 14GB Daten bereits 28,8 GB Indizes. Dies liegt daran, dass der Zugriff auf die aggregierten Daten wesentlich zeitkritischer ist, da diese für kundenseitig einsehbare Dienste genutzt werden. Daher enthalten diese Tabellen mehr Metainformation und mehr Indextabellen. So verfügen die voraggregierten Tabellen über je fünf Indizes, wogegen die realtime facts lediglich auf die id und den Zeitstempel indiziert werden.

Die Indizierung ist jedoch aus Skalierungssicht problematisch. Während sie für schnelle Zugriffe unabdingbar ist, bedeutet sie für Insert oder Update Operationen, dass zusätzlich die Indextabellen angepasst werden müssen, was zusätzlichen I/O-Overhead erzeugt. Zum anderen kostet die Indizierung auch sehr viel Platz. Insgesamt kommen auf die 24,1

GB Daten im Warehouse 34,9GB Indizes. Aufgrund dieser Größe kann die Datenbank nur einen Teil der Indizes im Cache halten. Damit verlangsamt sich auch der Lesezugriff weiter, da Indizes erst von der Festplatte gelesen werden müssen.
Somit ist zum einen Absehbar, dass einerseits die Indexgröße zunehmend zum Problem wird und andererseits die Skalierung dieser Lösung begrenzt ist, da relationale Datenbanken sich nur schwer horizontal skalieren lassen.

Weiteres Optimierungspotenzial ergibt sich aus dem Verzicht auf ein SQL basiertes System als solches und dem Umstieg auf eine schemalose Datenbank. SQL erzwingt im Schemadesign die Festlegung auf einen konkreten Datentyp für die gesamte Tabelle. So werden im aktuell genutzten Schema alle Datenpunkte unabhängig von ihrem nativen Datentypen als Double (64Bit) abgelegt. Auf die Summe der vorhandenen Daten gerechnet ist ein erhebliches Einsparpotenzial ersichtlich (Siehe Tabelle xy), wenn kleinere Typen nativ abgebildet werden könnten.

Typ	**Anzahl Tags**	**Größe in Bit**
String	96	-
Double	187	64
Boolean	23	1
Long	8	64
Integer	22	32
Float	56	32

Tabelle 2: Unterschiedliche Datentypen, die Häufigkeit ihres Aufkommens in als Tag und die Größe des Datentyps

Ebenso wird ersichtlich, dass Strings eine nicht zu verachtende Rolle in der Abbildung von Daten spielen. Diese können derzeit nicht im Warehouse abgebildet werden.

3.1.2 Anforderungen

Die Notwendigen Features für eine neue Warehouse-Lösung ergeben sich aus geschäftlichen und technischen Anforderungen der Firma Kiwigrid GmbH, sowie dem aktuellen Funktionsumfang des Warehouse.

Aus geschäftlicher Sicht ergeben sich die folgenden Anforderungen:

Anforderung	**HBase**	**OpenTSDB**	**MongoDB**
Hochverfügbarkeit	✓	✓	✓
Kapazitativ erweiterbar durch Hinzufügen weiterer Server	✓	✓	✓
Datenvergleich auf Geräteebene muss möglich sein	✓[1]	✓[1]	✓[1]

Tabelle 3: Geschäftliche Anforderungen an das Warehouse

Aus technischer Sicht ergeben sich die folgenden Anforderungen:

1 Dies ist in allen drei Systemen schemaabhängig

Anforderung	HBase	OpenTSDB	MongoDB
Hochperformant bei der Verarbeitung paralleler Anfragen (~ 8300 Inserts/s)	✓	✓	✓
Einführung von Echtzeitaggregation zur Erhöhung der Flexibilität (max 2s. Antwortzeit)	✓	✓	✓
Die Aggregation von Maximum, Minimum, Summe, Durchschnitt und Differenz über Zeitreihen muss möglich sein	✓[2]	✓	✓[3]
Die Abfrage der rohen Zeitseriendaten muss möglich sein	✓	×	✓
Aggregationen müssen mathematisch korrekt interpoliert werden	~[4]	×[5]	~[4]
Downsampling (Heruntertaktung) von Zeitreihen und Aggregationen muss unterstützt werden	✓	✓	✓
Die Einbindung von Batch-Operationen soll zur Bearbeitung großer Berechnungen ermöglicht werden	✓	✓	✓[6]

Tabelle 4: Technische Anforderungen an das Warehouse

2 HBase kann mit Hilfe des Aggregation Framework zwar aggregieren, jedoch nicht über Intervalle [vgl. HBase 0.98 APIDocs, AggregationClient]. Zu diesem Zweck wurde die TAggregator-Erweiterung entwickelt (Siehe 4.4).

3 Mit dem Aggregation Framework beherrscht MongoDB Aggregationen, die Intervallaggregation wurde jedoch Clientseitig realisiert.

4 Dies ist implementierungsabhängig

5 Die Interpolation in OpenTSDB ist sehr naiv. Die Bereiche vor- bzw. nach dem Intervall werden nicht berücksichtigt. [OpenTSDB Documentation, Interpolation]

6 MongoDB bietet zu diesem Zweck eine MapReduce Implementierung an. Für komplexe Analysen wird jedoch die Verwendung von Hadoop emfohlen [MongoDB Documentation, Hadoop Use-Cases].

3.2 Schema

Zur Entwicklung eines geeigneten Schemas ist ein grundlegendes Verständnis der gängigen Datenzugriffsmechanismen notwendig. Üblicherweise findet der Zugriff auf Zeitserien durch einen Portalnutzer statt. Dieser nutzt dabei vorgefertigte Statistikseiten um Informationen über Geräte abzurufen, die an einem ihm zugeordneten Energiemanager betrieben werden. Typischerweise werden dabei Dinge wie z.B. der Tagesverlauf in Stundenauflösung für den Gesamtverbrauch, sein Selbstversorgungsanteil, sowie sein Stromzukauf gemeinsam angezeigt. Diese Ansichten können dann weiter verfeinert werden, so dass eine Stunde minutenweise, ein Monat tagesweise oder das gesamte Jahr monatsweise dargestellt werden.
Hinzu kommen jedoch vermehrt Dashboardansichten, die eine Vielzahl von Absolutwerten und Verlaufskurven kombinieren. Dort werden derzeit teilweise Clientseitige Aggregationen genutzt, da das Warehouse die notwendigen Aggregationen nicht implementiert.
Gemeinsam haben diese Ansichten immer, dass Daten von Geräten abgerufen werden, die einem Nutzer zugeordnet sind. Diese Geräte sind dabei immer mittels einer GUID identifizierbar, sind selbst ein Energiemanager oder mit einem Energiemanager verknüpft und lassen sich als ein bestimmter Gerätetyp identifizeren.
Daraus ergibt sich, dass ein Datenpunkt über folgende Merkmale verfügen sollte:

- Tag – Identifikation des Werts (Einheit, Datentyp)
- Gerätetyp (z. B. SMA Wechselrichter) oder etwas abstrakter Geräteklasse (z. B. Wechselrichter)
- Energiemanager – Zuordnung einzelner Endgeräte zum übergeordneten Gerät
- GUID – Eindeutige Identifizierbarkeit eines Geräts und ggf. Zuordnnug zu Gerätetyp

Zusätzlich ist zu ermitteln, wie ein optimales Schema für die Speicherung von Zeitseriendaten in der Zieldatenbank umgesetzt werden kann. Dies ergibt sich hauptsächlich aus den Zugriffsmechanismen, die das System bietet und Beobachtungen, wie diese effizient eingesetzt werden können.

3.2.1 OpenTSDB Schema

Im Falle von OpenTSDB ist die Entwicklung eines Schemas nicht notwendig, da OpenTSDB dieses bereits vorgibt. Es muss jedoch eine Abbildung der in 3.2 erarbeiteten Identifizierungsmerkmale auf das vorhandene Schema entwickelt werden.
OpenTSDB sieht folgendes Schema vor:

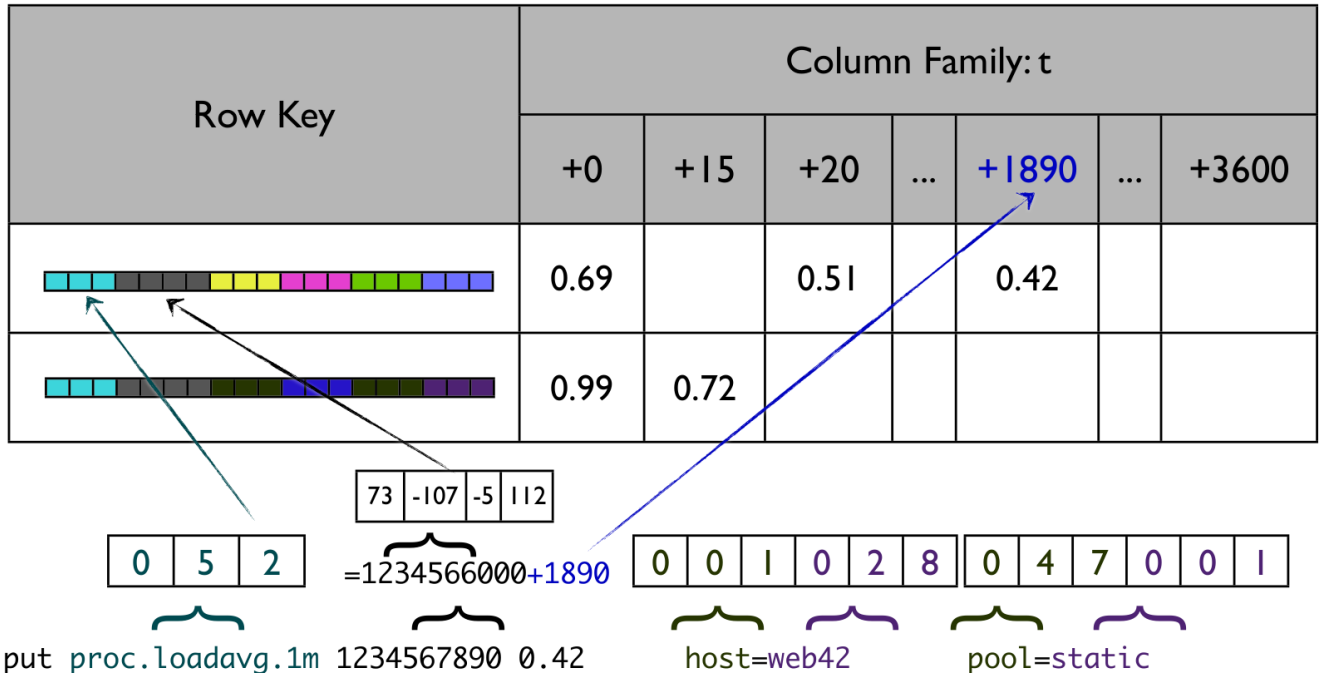

Illustration 2: OpenTSDB Schema [Sigoure, B. 2011. Slide 13]

Dabei wird zur Erstellung eines Key aus einem eindeutigen Identifikationskriterium einer Zeitserie eine 3-Byte lange ID gebildet. Diese wird mit dem Zeitstempel der aktuellen vollen Stunde konkateniert (4-Byte). Zusätzlich werden Metadaten angehängt (3 Byte pro Identifier). Diese Identifier werden in einer seperaten Zuordnungstabelle gehalten. Alle Werte, die dieser Zeitserie zu dieser vollen Stunde zugeordnet werden können, werden diesem Key zugeordnet. Dabei wird die Differenz zwischen dem genauen Zeitstempel und der vollen Stunde konkateniert mit dem Datentyp der Metrik als Column Qualifier in der Column Family „t“ gemeinsam mit dem Wert hinterlegt.
Im Falle der Kiwigrid GmbH wird der Key aus folgenden Eigenschaften gebildet:
die Metrik-ID wird aus dem Tag und der Geräteklasse konkateniert. Dadurch soll die Lokalität von Daten mit gleichem Tag, sowie gleicher Geräteklasse gewährleistet werden. Als zusätzliche Metadaten werden dann die GUID des Geräts, sowie der zugehörige EM mit gespeichert um eine eindeutige Identifikation bei Beibehaltung möglichst große

Flexibilität für die Vergleichbarkeit der Daten zu gewährleisten. So ist es mit diesem Schema sowohl möglich, einzelne Geräte abzufragen aber auch feingranular Geräte miteinander zu vergleichen. Die einzige Einschränkung besteht darin, dass lediglich gleiche Tags der gleichen Geräteklasse direkt verglichen werden können.

Dieses Schema bringt zahlreiche Vorteile mit sich:

- Sogenannte „Hot Spots“ werden durch das Prefixen des Timestamps mit der Metrik-ID vermieden. So werden Lese- und Schreiboperationen basierend auf der Metrik-ID im Cluster verteilt. [George, L. 2011, S. 366]
- Tabellen wachsen nicht nur vertikal, sondern durch die Speicherung vieler Werte unterhalb eines Keys auch horizontal an. Dadurch kann das I/O beim Lesen verbessert werden, da diese am Stück gelesen werden. [George, L. 2011, S. 330]
- Da der Key an jeder Spalte der Datenbank angehängt ist, kann durch das Ersetzen langer Strings durch Byte-Hashes sehr viel Platz eingespart werden (Sigoure, B. 2012. Slide 4). Dadurch ist es auch möglich, mehr Daten in Caches im Speicher vorzuhalten.
- Beim Schreiben wird Disk-Seek reduziert, da Stundendaten von Geräten am Stück weggeschrieben werden können [Sigoure, B., 2012 Slide 11].

3.2.2 HBase Schema

Bedingt durch die große Erfahrung in Sachen Schema-Design, welche die Macher von OpenTSDB durch die Entwicklung ihrer Lösung erarbeitet haben und die Tatsache, dass dieses Schema auch in der Literatur immer wieder als Beispiel für gutes Schema-Design herangezogen wird [George, L., 2011, S.366; Dimiduk, N., Khurana, A.,2013, 181ff.], wurde dieses ebenfalls als Grundlage für das entwickelte HBase-Schema genutzt. Es wurden lediglich kleine Änderungen vorgenommen, auf die ich im Folgenden eingehen möchte:

- Vergrößerung der Byte-Hashes auf 4 Byte zur Vergrößerung des Adressraums von 2^24 auf 2^32 mögliche Hash-Adressen
- Entfernen des Datentyps aus der Colum Family. Da alle Daten als Double vorliegen ist dies zunächst nicht notwendig. Auch zukünftig ist das Wissen über den Datentypen an dieser Stelle nicht notwendig, da es bereits aus dem Tag hervorgeht.

3.2.3 MongoDB Schema

Für MongoDB ist die Anwendung des OpenTSDB Schemas aufgrund der Ausrichtung des Schemas auf Dokumente so nicht möglich. Die Zeitseriendaten müssen auf Dokumente abgebildet werden, die dann indexiert werden können. In Anlehnung an den MongoDB MMS Dienst und das bereits vorgestellte Schema wurde jedoch beschlossen ähnlich vorzugehen [MongoDB Blog. Schema Design for Time Series Data in MongoDB]. So entstand folgendes Dokumentenschema:

Der Shard-Key wird analog zu HBase/OpenTSDB aus Hashwerten gebildet. Dadurch soll eine optimale Verteilung der Daten auf die Shards sichergestellt werden. Sämtliche Metadaten, die in diesem Key bereits vorliegen, werden außerdem als Klartext im Stundendokument abgelegt, um Indizes über diese bilden zu können. Die Zeitstempel-Wert Paare werden in dem „values“-Array des Dokuments abgelegt.
Durch die Speicherung des Vollständigen Zeitstempels zu jedem Wert ist es darüber hinaus möglich, das in MongoDB integrierte Aggregationsframework zu nutzen, sodass nicht Clientseitig aggregiert werden muss.

Nachteilig ist dabei allerdings, dass zur Speicherung eines Wertes das Array vollständig durchlaufen werden muss, jedoch kann so die Indexgröße stark reduziert werden, was den Zugriff beschleunigt. [MongoDB Blog. Schema Design for Time Series Data in MongoDB]

Ebenso versucht MongoDB nicht vorhandene Datendateien auf der Festplatte zu vergrößern, sondern, legt beim Schreiben auf eine vorhandene Datei eine größere Kopie der vorhandenen Datei an und beschreibt diese. Mit der Zeit versucht MongoDB einen sinnvollen Standardwert für die Dateigröße zu erlernen. [MongoDB Documentation. Record Allocation Strategies]

Während diese beiden Eigenschaften zu einer Verschlechterung des Durchsatzes führen können, führen sie andererseits zu einer Verkleinerung der indizes und zur Reduktion der Dokumentenanzahl, was sich insgesamt positiv auswirken dürfte.

3.3 Datensatz

Um möglichst praxisnahe Testdaten zu erhalten, wird zur Testdurchführung ein geeigneter Datensatz aus Einträgen der Produktivdatenbank erzeugt.
Zunächst wird dazu ein vollständier Auszug aus der realtime_facts Tabelle entnommen, der um „deleted_facts“ (Gerät, das Nutzerseitig bereits entfernt wurde, dessen Daten jedoch noch nicht gelöscht wurden) bereinigt wird. Nach der Durchführung dieser Bereinigung stehen noch 175.455.600 Datenpunkte zur Verfügung. Zusätzlich müssen diese Daten noch um die notwendigen Metainformationen erweitert werden, um das in 3.2 entwickelte Schema abbilden zu können.
So wurde aus mehreren MySQL Exports unter Verwenudng von MapReduce-Transformationen eine Textdatei generiert, die alle notwenidgen Daten nach folgendem Schema enthält:

Zeitstempel	GUID	Geräteklasse	Tag	Wert	EM

Da mit 175 Millionen Datenpunkten nur ein relativ kleiner Datensatz zur Verfügung steht, wird dieser nochmals durch eine um ein Jahr in die Zukunft verschobene Kopie eines jeden Datenpunktes erweitert. So stehen letztlich 350.911.200 Datenpunkte zur Verfügung.

3.4 Testkriterien

Um möglichst realitätsnahe Tests durchführen zu können, wird angestrebt, das Testsetup möglichst nah an den infrastrukturellen Gegebenheuten eines Produktivsystems anzulehnen. Das heißt, Daten werden im Datenbank-Cluster redundant gehalten und der Zugriff erfolgt von einem Client Multithreaded. Zukünfitg vorgesehen ist ein vorgelagerter Webservice, der Datenbankanfragen Multithreaded durchführt. Dies wird durch eine Clientapplikation, die auf einem Seperaten Server läuft, simuliert.

Wichtige Metriken für alle Tests sind die folgenden:

- Insertrate
- Lese- und Aggregationsgeschwindigkeit

Dazu kommen folgende, seperat durchgeführten Betrachtungen:

- Wie verhalten sich die Systeme in Fall von Fehlern (Ausfall eines aktiven Knotens,

fehlerhaft formatierte Daten)

- Ist die Balance der Daten im Cluster gegeben oder kommt es zu Hot-Spots

3.4.1 Testkriterien HBase

Mit HBase soll auf den Einsatz von Coprocessoren zur Datenbankseitigen Aggregation eingegangen werden. Zu diesem Zweck wurde der TAggregator, ein HBase-Coprocessor entwickelt, der einfache Aggregationen, wie die Berechnung von Maximum, Minimum, Summe und Durchschnitt auf Zeitserienbasis durchführen kann.

3.4.2 Testkriterien OpenTSDB

Die Leistungsfähigkeit der clientseitigen Aggregation, sowie die Leistungsfähigkeit des asynchronen Asynchbase sollen mit dem Thread Basierten HTable mit Coprocessor verglichen werden.

3.4.3 Testkriterien MongoDB

Es soll untersucht werden, ob und in wie weit das MongoDB Aggregationsframework für die Echtzeitaggregation von Zeitseriendaten geeignet ist. Zudem sollen die Unterschiede zwischen der Verwendung von B-Bäumen und LSM-Bäumen untersucht werden.

3.5 Einteilung der Testkriterien in Benchmarks

Folgende Benchmarks werden durchgeführt:

- Reine Messung der Insertrate
- Reine Messung der Leserate mit Aggregation
- Messung einer gemischten Last mit gleichzeitigem Lesen- und Schreiben von Daten

Gesondert betrachtet wird folgendes:

- Ausfall von Knoten beim Lesen- und Schreiben
- Datenbalance im Cluster

Bei der Messung der Insertrate werden die Daten in kleinen Batches von je 20 Datenpunkten geschrieben. Dies entspricht in etwa dem Datensynchronisationsverhalten eines Energiemanagers. Dabei wird eine unterschiedlich große Anzahl an Threads zum Schreiben verwendet. So soll ermittelt werden, wie das System unter Last skaliert.

Die Messung der Leserate ist in mehrere Szenarien unterteilt. Dabei wird zunächst die Leserate ohne vorhandene Schreiblast getestet. Auch hier, wird die Last über mehrere Testläufe durch eine Vergrößerung der Threadanzahl erhöht. In einem weiteren Schritt soll das synchrone Lesen- und Schreiben simuliert werden.

Anfragen für das Lesen werden automatisch generiert. Die Anfragen folgen den folgenden Regeln:

- Eine der folgenden Aggregationsfunktionen muss gesetzt sein:
  - Max – Bildung des Maximums
  - Min – Bildung des Minimums
  - Sum – Bildung der Summe
  - AVG – Bildung des Durchschnitts
  - NONE – Ausgabe der Zeitseriendaten ohne Aggregation
- Eine zufälliger Zeitbereich mit folgendem Umfang muss gesetzt sein:
  - Jahr
  - Monat
  - Woche
  - Tag
  - Halber Tag
  - Stunde
- Zusätzlich zum Zeitbereich muss ein Aggregationsintervall gesetzt sein, welches kleiner als der Bereich sein muss:
  - Monat – darf für Jahre aggregiert werden
  - Woche – darf für Monate und Jahre aggregiert werden
  - Tag – darf für Jahre, Monate und Wochen aggregiert werden
  - Halber Tag – darf für Monate aggregiert werden

    - Stunde – darf für Wochen, Tage und halbe Tage aggregiert werden
    - Viertelstunde – darf für Wochen, Tage und halbe Tage aggregiert werden
    - Fünf Minten – darf für Tage, halbe Tage und Stunden aggregiert werden
    - Minute – darf für Stunden und halbe Tage aggregiert werden
- Eine Anfrage enthält immer eine Kombination aus Tag und DeviceClass
- Eine Anfrage enthält immer einen EnergieManager
- Eine Anfrage muss mindestens eine GUID enthalten

Beispielqueries aus diesem Pool sind z. B. eine Abfrage der maximalen Ausgangsleistung von 3 Invertern eines EMs über ein Jahr in Wochenintervallen oder die Abfrage der Maxima eines Tages an Zählerwerten eines Zählers in Minutenintervallen.
Durch diese Regeln ist es möglich, ein relativ weites Feld an Anfragen zu stellen, komplexere Anfragen, die z. B. über mehrere Energiemanager gehen bleiben so jedoch außen vor. Dabei ist jedoch anzuzweifeln, ob bei einer solchen, zufälligen Abfrage über mehrere Energiemanager überhaupt zwei Geräte gefunden würden, die zur gleichen Zeit Daten aufgezeichnet haben.

4. Testdurchführung

Zur Durchführung der Tests wurden verschiedene Skripte, Benchmarktools und weitere Komponenten entwickelt. In diesem Abschnitt soll auf die Methodik eingegangen werden, die diesen Testwerkzeugen zugrunde liegt sowie deren Implementierung beleuchtet werden. Abschließend wird auf die Konfiguration der getesteten Systeme eingegangen, anhand derer sich diese Tests reproduzieren lassen.

4.1 Methodik

Um den Einfluss externer Faktoren wie vorübergehende Netzwerk-Congestion oder Festplattenfehler gering zu halten, werden alle Tests mindestens drei mal durchgeführt.
Nach jedem Test werden die Systeme neu gestartet, um zu verhindern, dass bei den folgenden Läufen ein Vorteil durch Caching entsteht.
Von den gewonnenen Messungen werden im weiteren Verlauf die Höchste und die Niedrigste nicht weiter betrachtet. Bei den Java basierten Systemen HBase und OpenTSDB werden darüber hinaus zunächst mehrere Probeläufe durchgeführt, damit die JVM Laufzeitoptimierungen vornehmen kann, die sonst mit der Zeit auch den Betrieb beschleunigen würden [vgl. Hawkins, D. 2010].

Bei den schreibenden Tests wird alleine die Dauer des Inserts gemessen. Da die Daten bereits in einem wohl definierten Format vorliegen und sie lediglich in kleinen Batches von bis zu 20 Datenpunkten geschrieben werden, ist hier der Datentransformationsaufwand in ein schreibbares Format vernachlässigbar. Darüber hinaus wird versucht, Verzögerungen durch I/O, die zu einer Verfälschung des Ergebnisses führen würden, durch einen Lesecache und vorzeitigen Abbruch im Falle des Leerlaufens des Caches zu vermeiden.

Bei den lesenden Tests wird im Gegensatz zu den schreibenden nicht lediglich die Dauer des Requests, sondern auch die Transformation in einen aus Java nutzbaren Datensatz gemessen. Die Begründung dafür liegt einerseits darin, dass in MongoDB nochmals ein clientseitiges Zusammensetzen der einzelnen Bereichsanfragen notwendig ist, zum anderen aber auch darin, dass jedes der verwendeten Systeme ein anderes Übertragungsformat verwendet, welches dann noch in eine benutzbare Form transformiert

werden muss. Für die Anfragen wird auf ein zufällig generiertes Set an Queries gesetzt. Der Anteil an Anfragen zu denen tatsächlich Daten in der Datenbank vorhanden sind (valide Anfragen) liegt bei 12%. So kann zusätzlich zu den validen Anfragen eine Grundlast simuliert werden. Auch aktuell ist es so, dass häufig Lücken in Zeitreihen vorkommen. Viele Wechselrichter schalten sich z. B. über nach ab, wodurch Lücken entstehen. Es ist also auch durchaus gewollt, solche Anfragen als Teil der Testmenge zu haben. Zudem wird so verhindert, dass repetitiver Zugriff zu einer Beantwortung aus dem Cache führt.

In einem weiteren Testszenario sollen diese beiden Tests zusammengeführt werden. Dabei wird das gleiche Leseszenario wiederholt, während fünf Prozesse zeitgleich Werte in die Datenbank schreiben. Diese neuen Werte liegen in der Zukunft, um zu verhindern, dass in bereits vorhandene Datenbereiche geschrieben wird. So soll ermittelt werden ob diese beiden Prozesse Schreiben und Lesen sich gegenseitig beeinflussen.

4.2 Abgrenzungen

Für die Betrachtung von Zeitseriendaten in einem Produktivsystem ist es notwendig, auch Zeitzonen sowie Daylight Savings Time (DST) zu beachten. Für die hier durchgeführten Tests wird dies nicht als relevant erachtet. Es ist jedoch neben der Reduzierung von Disk Seek ein Grund dafür, dass Daten Stundenweise gebündelt werden, da so der Stundenweise Versatz der DST später abgebildet werden soll.
Auch wurde davon abgesehen, Daten in ihren nativen Typ zu überführen, da dies den Vorbereitungsaufwand des Datensatzes erheblich gesteigert hätte und es zur reinen Leistungsevaluierung als nicht relevant angesehen wird.
Auf Messungen bezüglich des Einflusses von Kompression wird ebenfalls verzichtet, da dies bereits hinreichend in anderen Quellen dokumentiert wurde [Soboroff, I, 2013; Kamat, G., Singh, S. 2013, S. 8; Comerford, A., 2014; MongoDB Blog, 2014].

4.3 Systemkonfiguration

Für das Testsetup wird aus praktischen Gründen auf Virtualisierung gesetzt, da sonst bis zu zwölf Server für die Testdurchführung angeschafft werden müssten. Als Hostsystem kommen drei Hetzner EX4 [Hetzner Wiki, EX 4 Server] im gleichen Rechenzentrum zum

Einsatz. Diese sind mittles einer separaten Netzwerkschnittstelle über ein Internes GBit-Ethernet Netzwerk untereinander verbunden (maximal 2 Hops).

Komponente	**Spezifikation**
Prozessor	Core i7-2600; 3,4 Ghz (3,8 Ghz Turbo); 4 Kerne mit Hyperthreading (8 virtuelle Kerne)
Arbeitsspeicher	16GB DDR3
Festplatte	2x 3TB Seagate, Toshiba; MD/RAID 1
Netzwerk	Intel PRO/1000 (intern) Realtec RTL8168e/8111e (extern)

Tabelle 5: Hardwarebestückung der Server

Als Betriebssystem kommt auf allen Servern Debian GNU/Linux in der Version 7.0 (Wheezy) zum Einsatz. Das Dateisystem ist ext4. Die Virtualisierung wird mit Hilfe der Containervirtualisierung (auch Betriebssystemvirtualisierung genannt) OpenVZ umgesetzt. Dabei handelt es sich um eine leichtgewichtige Virtualisierungslösung, die keine Hardwareunterstützung oder Hardwareemulation benötigt. Die Container nutzen den Kernel des Hostsystems mit, es geschieht lediglich eine Ressourcenisolation auf Prozessebene. Dies bringt für die Überwachung der Benchmarks kleine Nachteile mit. So ist es nicht möglich, das Festplatten I/O Containergenau zu bestimmen. Jedoch überwiegen die Vorteile. So stehen kernelseitige Optimierungen automatisch in allen Containern zur verfügung. Auch sind Setup- und Deployment schneller, da Container ohne Installation aus einem Template gestartet werden können. Als wichtigstes Argument ist jedoch anzuführen, dass der I/O Overhead von Containern sehr klein ist. Dadurch, dass alle Container auf den gleichen Kernel setzen und der Hypervisor als zusätzliche Abstraktionsschicht fehlt, sind weniger Kontextwechsel für den Hardwarezugriff notwendig, was zu einer signifikant besseren Performance als bei einer Hypervisor-basierten Virtualisierung führt [vgl. Xavier, M. et. al. 2013. S. 5].

Weiterhin ist es wichtig, dass die Testsysteme in ihrer Konfiguration mit einem Produktivsystem vergleichbar sind, um möglichst viele Aspekte der zu testenden Systeme kennenzulernen und abschätzen zu können, ob diese für den Produktivbetrieb geeignet sind, bzw. um zu erfahren, welche besonderen Eigenschaften im Produktivbetrieb zu beachten sind.

Zunächst wurde eine Reihe von Kerneloptimierungen vorgenommen, die von Distributoren und großen Nutzern von HBase/MongoDB vorgeschlagen werden. Folgende Optimierungen wurden auf den Hostsystemen vorgenommen:

Parameter	**Beschreibung**
net.ipv4.ip_forward=1	IPv4 Forwarding – notwendig für die Netzwerkkommunikation der OpenVZ-Container
kernel.sysrq=1	“Magic SysRQ” - Flag, das die Ausführung bestimmter Kommandos im Fall einer Kernel Panic erlaubt. Sehr hilfreich zum Debugging
kernel.core_uses_pid=1	Aktiviert Core File Support zu Debugging zwecken
kernel.pid_max=4194303	Erhöht die Maximal mögliche Prozessanzahl - Diese zu erhöhen ist für Hadoop empfohlen
fs.file-max=204708	Erhöht die Maximal mögliche Anzahl offener File Handles – Dies ist empfehlenswert um das ulimit erhöhen zu können, was für MongoDB empfohlen wird
fs.epoll.max_user_instances=4096 (4096, da 2 Knoten pro Server)	Erhöht das Prozess/Nutzer Limit für Epoll, ein Mechanisum zum Beobachten von File-Handles – Das Standardlimit von 128 wird schnell von Hadoop ausgeschöpft, weshalb eine Erhöhung auf 2048 empfohlen wird. [vgl. Hadoop Wiki, TooManyOpenFiles]
net.core.somaxconn=4096	Erhöht die Anzahl maximal möglicher wartender Socketverbindungen – empfohlen für den Hadoop/HBase Netzwerkverkehr [Gogate, V. 2014. Slide 15]
net.ipv4.tcp_moderate_rcvbuf=1	TCP Receiver Autotuning- passt den TCP Receive Buffer automatisch an die Netzwerkschnittstelle an um den Durchsatz zu maximieren
net.ipv4.tcp_fastopen = 3	Schnelles öffnen von Unix Domain Sockets – Hadoop-6311 [Apache Issue Tracker. HADOOP-6311]

Tabelle 6: Veränderte Kernelparameter und deren gesetzter Wert

Da die Container aus Sicht des Hosts lediglich isolierte Prozessbäume sind und daher den Host-Kernel mitnutzen, werden diese Optimierungen auch automatisch für die Container wirksam.

Alle Systeme sind in das Monitoringsystem Zabbix Integriert, um Ausfälle von Knoten, einzelnen Hardwarekomponenten, wie Festplatten oder Diensten frühzeitig erkennen zu können.

4.3.1 HBase

Installiert werden sechs Knoten auf Basis von CentOS6 mit je vier virtuellen Kernen, 6GB Ram und 300GB HDD. Auf jedem physischen Server sind je zwei Container deployed. Für eine detailierte Auflistung, siehe Appendix I.

Für den produktiven Betrieb wird mindestens eine dreifache Datenreplikation empfohlen. In seiner Standardkonfiguration ist es sogar so, dass HDFS mit einem geringeren Replikationsfaktor in den Betriebsmodus Safe-Mode verweilt und das Schreiben von Daten verweigert. Eine dreifache Replikation kann ab drei Knoten gewährleistet werden, ist also in diesem Setup gegeben.
Mit Hilfe der installierten ZKFailoverController wird darüber hinaus Hochverfügbarkeit für die HDFS NameNodes (hdp0, hdp5) gewährleistet.

Die zum Einsatz kommenden Softwareversionen entsprechen der Hortonworks Data Platform (HDP) 2.1, welche vom verwendeten Ambari 1.6.1 als Paketquelle genutzt wird.

Service	**Version**	**Beschreibung**
HDFS	2.4.0.2.1	Apache Hadoop Distributed File System
YARN + MapReduce2	2.4.0.2.1	Apache Hadoop NextGen MapReduce (YARN)
Tez	0.4.0.2.1	Tez ist ein Anfrageverarbeitungsframework auf der Basis von YARN.
Nagios	3.5.0	Das Nagios Monitoring und Alerting System
Ganglia	3.5.0	Das Ganglia Metriksystem
HBase	0.98.0.2.1	Eine nicht relationale, verteilte Datenbank
Pig	0.12.1.2.1	Eine Skriptsprache/Plattform für die Analyse großer Datensätze
ZooKeeper	3.4.5.2.1	Ein zentralisierter, hochverfügbarer Cluster-Koordinierungsdienst

Tabelle 7: Installierte Komponenten des Hadoop Stacks und deren Version

Zusätzlich wurde für die Verwendung mit HBase und OpenTSDB die HDFS LZO Native Extension installiert. Dabei handelt es sich um ein Modul, welches die Nutzung von LZO-Kompression in HDFS – und damit auch in HBase – erlaubt.

Darüber hinaus wurde in HBase HBASE-5175 [Apache Issue Tracker. HBASE-5175] nachgepatcht, um Double-Werte interpretieren zu können. Dieses Feature ist standardmäßig erst ab HBase 0.98.1 verfügbar.

Der eigens für die hier durchgeführten Tests entwickelte TAggregator (Siehe 4.4) wurde ebenfalls nachinstalliert.

4.3.2 OpenTSDB

Da OpenTSDB das vorhandensein von HBase voraussetzt, nutzt OpenTSDB die in 2.7.1 beschriebene HBase Installation. Nach der Erzeugung der für OpenTSDB notwendigen Tabellen sowie der Konfiguration des Zugriffs ist OpenTSDB sofort einsetzbar. Die für den Zugriff notwendigen TSDs wurden zusätzlich auf zwei vorhandenen HBase Knoten installiert:

Knoten	**Installierter OpenTSDB-Dienst**
hdp3.int.kiwidev.com	1x TSD
hdp4.int.kiwidev.com	1x TSD

Tabelle 8: OpenTSDB Knoten

Es kam die OpenTSDB Version 2.0.0 zum Einsatz. Die verwendete HBase Version ist 4.3.1 zu entnehmen. Eine Beschreibung des HBase Setup ist in Appendix I zu finden.

4.3.3 MongoDB

Aufgrund der Menge an benötigten Containern für Replication Groups, Config Server und Shards ist ein Setup mit sechs Shards, wie es für HBase gewählt wurde, mit der vorhandenen Hardware nicht praktikabel, ohne vollständig auf Replikation zu verzichten. Da ein Setup ohne Replikation nicht zu einer Produktivumgebung repräsentativ ist, wurde ein Mittelweg unter Verwendung von Arbitern gewählt:

Knoten	**Installierte Anwendungen**
mng1a1.int.kiwidev.com (Server 2)	• Replica Set 1 Master
mng2a1.int.kiwidev.com (Server 1)	• Replica Set 1 Slave
mng3a1.int.kiwidev.com (Server 3)	• Replica Set 1 Arbiter • Config Server • Mongos Router
mng1a2.int.kiwidev.com (Server 1)	• Replica Set 2 Master
mng2a2.int.kiwidev.com (Server 3)	• Replica Set 2 Slave
mng3a2.int.kiwidev.com (Server 2)	• Replica Set 2 Arbiter • Config Server • Mongos Router
mng1a3.int.kiwidev.com (Server 3)	• Replica Set 3 Master
mng2a3.int.kiwidev.com (Server 2)	• Replica Set 3 Slave
mng3a3.int.kiwidev.com (Server 1)	• Replica Set 3 Arbiter • Config Server • Mongos Router

Tabelle 9: Installierte MongoDB Knoten

Aus Tabell 9 wird ersichtlich, dass bei der Replikation Abstriche gemacht wurden und lediglich auf eine zweifache Replikation gesetzt wird. Dies liegt darin begründet, dass auf zusätzliche Container verzichtet werden sollte, um genug freie Ressourcen für den Container-Host zur Verfügung stellen zu können und den Mongod Servern dennoch ausreichend Arbeitsspeicher zur Verfügung stellen zu können. Darüber hinaus ist die größere Shardanzahl wichtig, um einen besseren Vergleich zur Skalierbarkeit von HBase treffen zu können.

Installiert werden neun Knoten auf Basis von CentOS6. Pro Knoten je zwei mit vier virtuellen Kernen, 7GB Ram und 300GB HDD und ein Knoten mit einem Kern und 512MB Ram als Arbiter.

Zusätzlich wurden, nachdem diesbezügliche Fehler auftraten, auf jedem MongoDB Container die Pro-Nutzer File-Handle und Prozesslimits für den „mongod“-Nutzer, unter dem MongoDB läuft, von 1024 auf 32.000 erhöht. Dabei ist zu beachten, dass es sich in diesem Fall nicht um vom Kernel gesetzte Limits handelt, sondern um Limits, die von CentOS aus Sicherheitsgründen gesetzt werden. Diese müssen sich jedoch im Rahmen der globalen, vom Kernel bestimmten Parameter bewegen [vgl. MongoDB Documentation. Ulimit].

Die Tests wurden zunächst mit MongoDB Version 2.6.4, dem aktuellen stabilen Release begonnen. Im Laufe der Tests wurde jedoch die Entscheidung getroffen, auf Version 2.8.0 Release Candidates umzuschwenken. In Version 2.6.4 wurden schnell Probleme im Locking offensichtlich, die so nicht lösbar waren und zu extrem schlechter Performance im Sharding führten. Version 2.8 enthält eine neue Storage Engine namens WiredTiger, die nun ein sehr feingranulares Locking auf Dokumentenebene sowie Kompression erlaubt, wogegen die in Version 2.6 zur Verfügung stehende MMAPv1 Engine lediglich auf Collection-Ebene sperrt. Da dadurch nicht mehr bei jeder Schreiboperation die Collection über den gesamten Cluster gelockt wird, wurde eine signifikante Verbesserung der Performance erwartet.

4.4 Implementierung

Zur Durchführung der Tests wurden mehrere Werkzeuge in Java Implementiert. Diese bilden jeweils das Lesen, bzw. Schreiben auf alle zu testenden Systeme in gleicher Form ab.

kiwigrid.warehousewritetest

Dieses Programm wurde entwickelt, um das Schreiben von Daten in die Testsysteme abzubilden. Es ermöglicht das Schreiben auf HBase, OpenTSDB, sowie MongoDB.
Dazu nutzt es den HBase Java-Client in Version 0.98.5, den MongoDB Java-Client in Version 2.12.3, sowie den Apache HTTPClient in der Version 4.3.5 für den Zugriff auf OpenTSDB.

Für die Messungen erfasst es in jedem schreibenden Thread die Dauer der Batch-Anfragen in ms sowie die Gesamtdauer. Dadurch wird die durchschnittliche Dauer eines Batch Inserts, wie auch die Gesamtdauer erfasst.

kiwigrid.warehousereadtest

Dieses Programm wurde zum Testen der Leserate geschrieben. Es wurde zur Ausführung der automatisch generierten Datenbankanfragen entwickelt. Wie der kiwigrid.warehousewritetest, erlaubt es das Lesen aus allen getesteten Datenbanksystemen. Dabei wird jedoch lediglich die Dauer, Art und Anwort der Anfragen aufgezeichnet. Es kann also lediglich die Dauer der einzelnen Anfragen erfasst werden. Es ist jedoch später nachvollziehbar, welche Anfragen überhaupt Ergebnisse produziert haben und welche nicht.

Kiwigrid.querybuilder

Ein Werkzeug zum Generieren von Datenbankanfragen. Es erzeugt 20.000 zufällige Datenbankanfragen in einem für den kiwigrid.warehousereadtest kompatiblen Eingabeformat.

TAggregator

Der TAggregator ist ein eigens entwickelter HBase Coprocessor. Im Gegensatz zu dem in HBase enthaltenen AggregationClient beherrscht dieser die Aggregation über Zeitserien. Sein Interpolationsvorgehen ist dabei gleich der vorgehensweise in OpenTSDB naiv. Die größe von Lücken und Intervallgrenzen werden auch hier nicht genauer betrachtet. Dies erschien als zu aufwendig für den Rahmen der Evaluation. Der TAggregator wurde unter Apache License Version 2 veröffentlicht und ist unter https://github.com/juwi/HBase-TAggregator veröffentlicht.

4.5 Testdurchführung

Die Durchführung erfolgt von zwei Rechnern aus, die sich im gleichen Netzwerk wie der Cluster befinden. Sie sind mit Gbit Ethernet mit dem Cluster verbunden und können jeden Knoten im maximal zwei Hops erreichen. Diese werden zur Ausführung der Schreib- und Lesetests genutzt.

Die meisten Tests konnten ohne weitere Probleme ausgeführt werden und erzielten in etwa die erwarteten Ergebnisse. Jedoch gab es auch immer wieder Schwierigkeiten, die im Folgenden erläutert werden.

Während der Ausführung kam es immer wieder zu Problemen mit einzelnen Festplatten, wodruch letztlich durch den notwendigen Hardwareaustausch immer wieder längere Pausen zwischen den Tests entstanden, während der Raid neu gesynct wurde.

Leider traten auch im Laufe der Tests mit MongoDB immer wieder große Probleme mit dem Region Balancing auf, wodurch letztlich keine Tests mit der B-Tree Engine durchgeführt werden konnten. Das Problem ist auf den Issue SERVER-15691 zurückzuführen [vgl. MongoDB Issue Tracker. SERVER-15691]. Dabei bleibt das Balancer Lock auf einem Datenbankknoten hängen. Die Verteilung von Daten ist daraufhin nicht mehr möglich. Dieses Verhalten ist sowohl mit der aktuell Stabilen Version 2.6.5, wie auch allen bisher erschienen Release-Candidate Version des 2.8 Zweigs reproduzierbar. Zunächst bestand Hoffnung, dass das Problem bis zum Erscheinen der Version 2.8-RC3 gefixt ist, jedoch wurde ein Fix bisher mehrfach verschoben und es ist nicht absehbar, bis

wann eine Lösung zur Verfügung steht.

Daneben scheint es in MongoDB auch Probleme mit parallelen Upserts zu geben, wenn beim gleichzeitigen Insert zweier Werte aus der gleichen Zeitserie festgestellt wird, dass ein neues Dokument angelegt werden muss. Dabei kam es im Laufe der Tests mehrfach dazu, dass MongoDB parallel versuchte, zweimal das gleiche Dokument neu anzulegen. Dieses Verhalten ließ sich nicht gezielt reproduzieren, führte aber bei jedem Auftreten zum Crash des mongod-Prozesses, der die Anfragen ausführte. Schlimmstenfalls konnte dabei dann das Write-Lock nicht mehr ergriffen werden, da es noch von dem abgestürzten Prozess gehalten wurde, was zu einem Stillstand der Inserts führte. Wird dies nicht sofort behoben, kann die Folge Datenverlust sein.

Auch mit OpenTSDB traten im Verlauf der Tests immer wieder kleinere Probleme auf. So kam es beim Lesen von Daten immer wieder zu Exceptions auf dem OpenTSDB Server, die darauf hinweisen, dass Asynchbase noch nicht ganz reibungslos mit HBase 0.98 kommunizieren kann, obwohl diese Version laut den Entwicklern ab OpenTSDB Version 2.0.0 unterstützt wird [Sigoure, B. 2014. S. 37]:

```
Unexpected exception from downstream on [id: 0x17e2cca6, /192.168.11.108:57790 =>
/192.168.11.107:60020] org.hbase.async.InvalidResponseException:
Should not have gotten any cell blocks, yet there are 399 bytes that follow the protobuf response.
This should never happen. Are you using an incompatible version of HBase?
```

Ein besonders großes Problem daran ist, dass dadurch die Beantwortung der HTTP-Anfrage unterbrochen wird, was eine extrem schlechte Praxis ist. So kann die Verbindung nicht unmittelbar geschlossen werden, sondern bleibt bestehen bis ihr Timeout erfolgt und es ist Clientseitig nicht mehr nachvollziehbar was passiert ist.
Dieses Verhalten ließ sich ebenfalls beobachten, wenn eine Metrik-ID abgefragt wird, die nicht existiert. Auch hier erfolgt keine Antwort des TSD.

Darüber hinaus wurde festgestellt, dass das Setzen der abzufragenden OpenTSDB Tags (EnergieManager und GUID) in der API-Anfrage reproduzierbar dazu führt, dass

OpenTSDB keinerlei Daten findet. Die Ursache für dieses Fehlverhalten konnte nicht ermittelt werden.

5. Ergebnisanalyse

5.1 Testauswertung

5.1.1 Beschreiben der Datenbanken

Beim Beschreiben ergibt sich ein recht eindeutiges Bild zur Leistungsfähigkeit [Illustration 3].

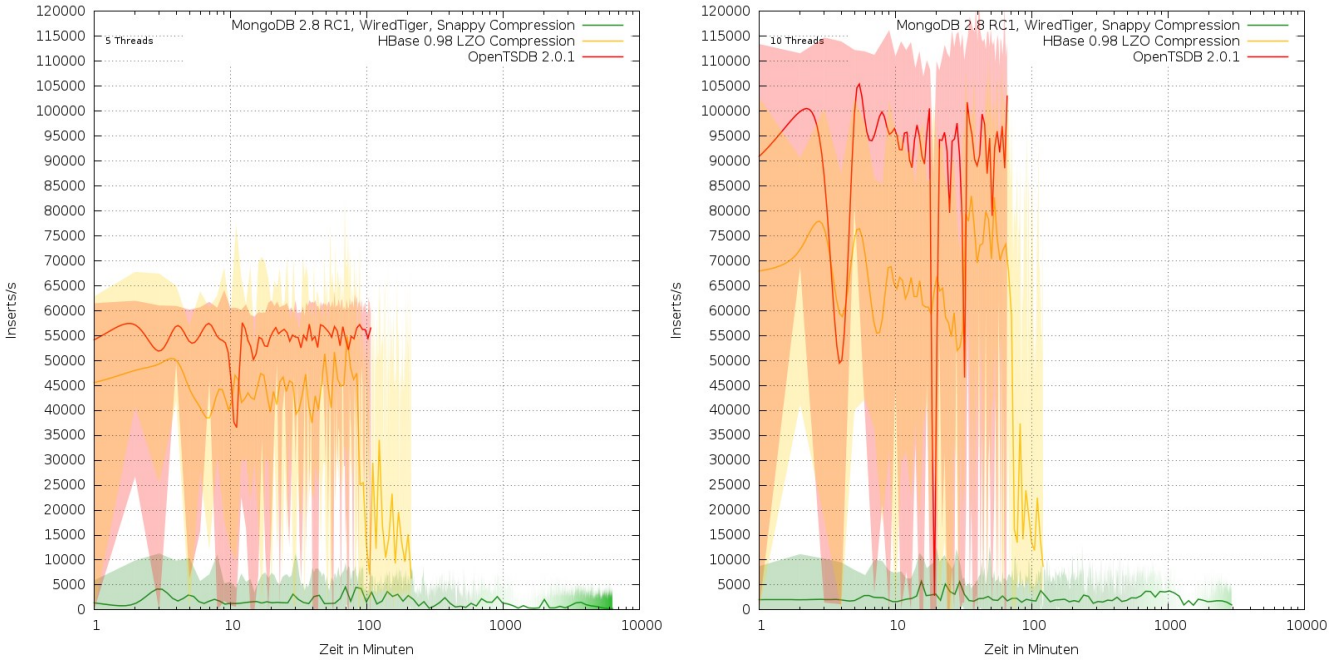

Illustration 3: Inserts Absolut

Die durchschnittliche Insertgeschwindigkeit von HBase bewegt sich bei fünf, wie auch bei zehn Threads zwischen 3,6 und 4,1ms. Die sichtbaren Peaks entstehen zu Zeitpunkten, wo auf dem HBase Server Region Migrations stattfinden. Dabei wird eine Datenregion auf einen anderen Server umgezogen. OpenTSDB ist mit einer durchschnittlichen Latenz zwischen 1,75ms und 2,15ms sogar noch etwas schneller. Man erkennt, dass dieser mit Region Splits besser umgeht. Zudem fehlt der Overhead, der sich durch die Verwendung eines Threadpools auf Clientseite ergibt. MongoDB schneidet hier sehr viel schlechter ab. Durch die extremen Peaks ergeben sich in beiden Fällen durchschnittliche Insertzeiten von ca. 100Ms, die sicherlich zum großen Teil im verteilten Locking auf dem Cluster liegt. Dieses Bild verstärkt sich nochmals, wenn man die absolut benötigte Zeit für alle Inserts betrachtet [Illustration 4].

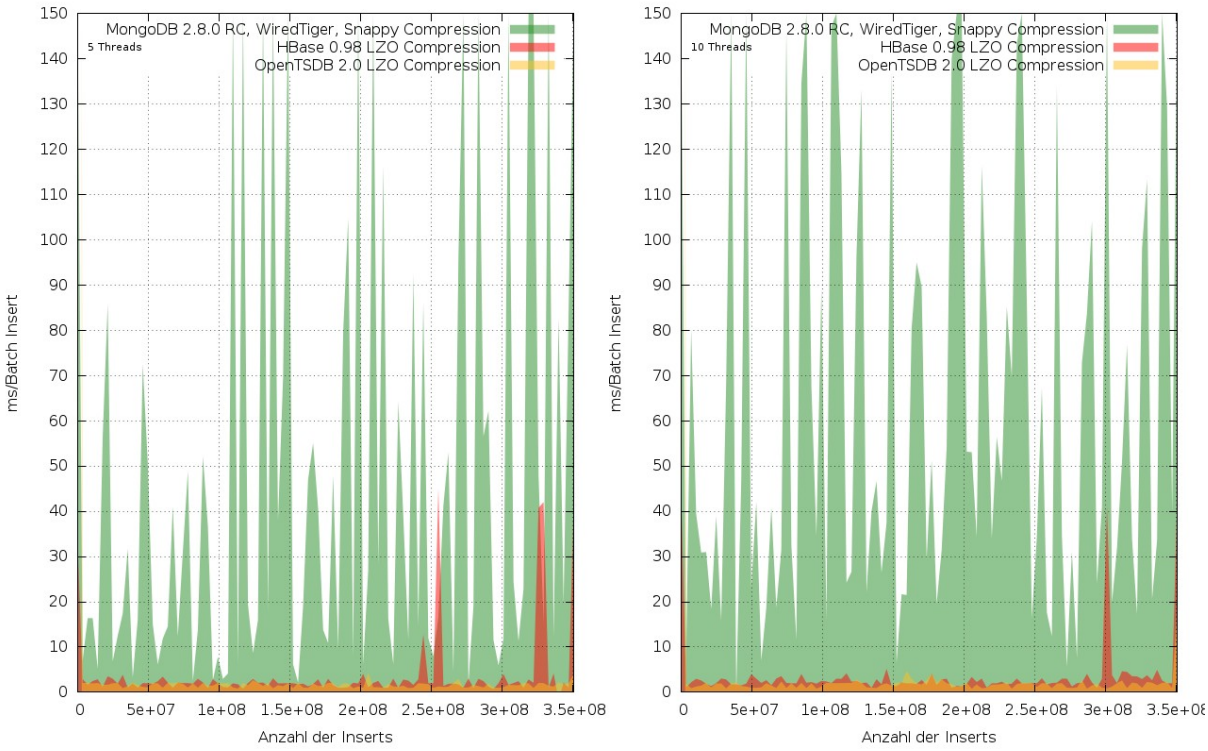

Illustration 4: Durchschnittliche Insertdauer

Hier fällt insbesondere die enorme Leistungsfähigkeit von OpenTSDB ins Auge. Dabei stellen die volltönigen Linien die durchschnittliche Insertrate dar, die eingefärbten Bereiche Maxima und Minima. Die starken negativen Peaks bei OpenTSDB und HBase korrellieren auch hier mit den Zeitpunkten der Region Splits. Interessant ist jedoch auch die starke Streuung der Ergebnisse. Zwar ist eine gewisse Streuung normal, in diesem Extrem könnte sie jedoch auch ein Nebeneffekt der verwendeten Virtualisierung und des daraus resultierenden stärkeren Disk-Seek sein, welcher durch die Replizierung der Daten in HDFS als Nebeneffekt entsteht.

Beeindruckend ist jedoch der Performancezuwachs von OpenTSDB, wie auch HBase, der aus der Verdoppelung der vorhandenen Insert-Threads resultiert. Die Leistung steigt bei beiden Systemen nochmals beträchtlich, wenn 20 Threads verwendet werden. Damit ließ sich jedoch der Test nicht vollständig durchführen, da dann die Daten nicht mehr schnell genug gelesen werden konnten. Dies lässt jedoch darauf schließen, dass sowohl HBase, wie auch OpenTSDB in dem vorhandenen Setup keineswegs durch die Hardware, sondern hauptsächlich durch die Ausführungsgeschwindigkeit der Threads selbst limitiert wird.

OpenTSDB

Betrachtet man den Festplattendurchsatz fällt auf, dass sich der Festplattendurchsatz auf allen drei Systemen recht gleichmäßig bei unter 30 MB/s bewegt [Illustrationen 5, 6, 7]. Im Lichte dessen, dass HBase Datenblöcke erst beim Erreichen eines 256MB Limits auf die Festplatte geschrieben werden, das Log des LSM-Tree ebenfalls sequentiell geschrieben wird und dass die Fragmentierung des Dateisystems bei 0,1% liegt, kann mit Sicherheit davon ausgegangen werden, dass kein Flaschenhals durch die Festplatten entsteht. Sequentielles Schreiben sollten diese mit bis zu 120 MB/s bewältigen.

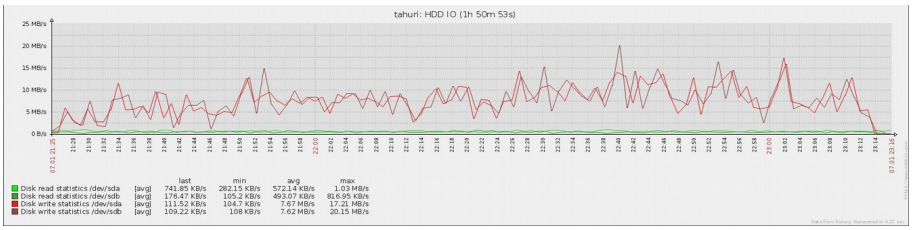

Illustration 5: OpenTSDB Festplattendurchsatz Tahuri 5 Threads

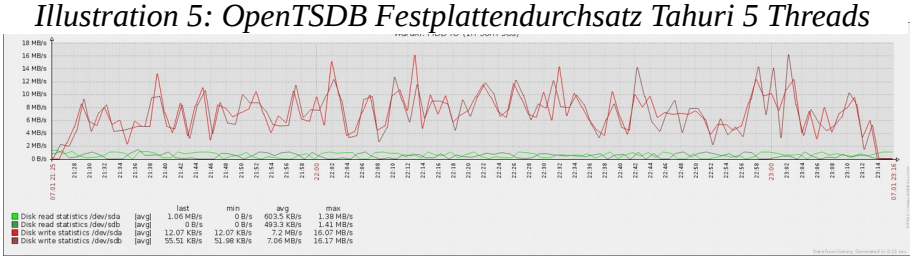

Illustration 6: OpenTSDB Festplattendurchsatz Waraki 5 Threads

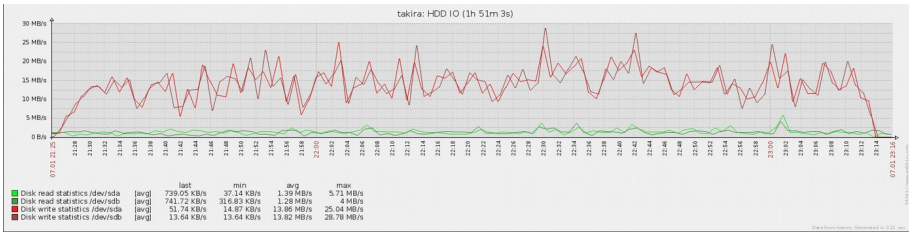

Illustration 7: OpenTSDB Festplattendurchsatz Takira 5 Threads

Beim Netzwerkdurchsatz [Illustration 8] zeigt sich wenig überraschend die Hauptlast auf dem Knoten hadoop108. Auf diesem läuft der TSD. Im Vergleich zu dem von HBase und MongoDB verwendeten RPC Protokoll ist dieses mit einem gewissen Protokolloverhead behaftet. Durchschnittlich wird mit 88Mbits an den TSD übertragen, dieser überträgt dann mit durchschnittlich 76 Mbits an HBase weiter. Hier besteht also ein nicht bedeutender Overhead gegenüber dem HBase RPC-Protokoll. Dieser Effekt verstärkt sich bei der Verwendung von zehn Threads nochmals. Da die theoretisch maximale Übertragungsleistung jedoch bei 1Gbits liegt, ist hier in beiden Fällen noch ein

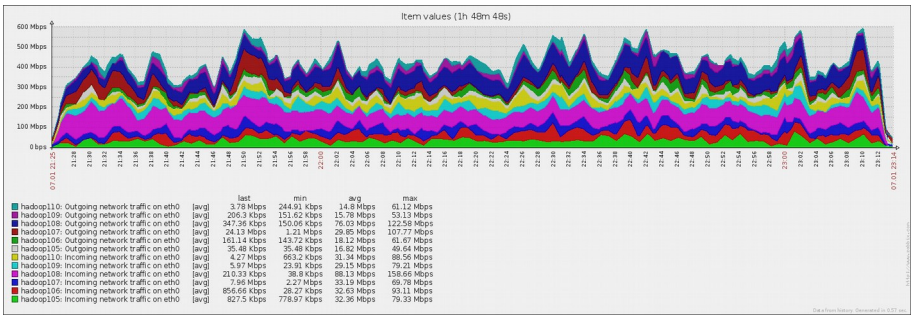

Illustration 8: OpenTSDB Netzwerkdurchsatz; Inserts mit 5 Threads

komfortables Polster vorhanden.

Betrachtet man die Load Average, so ergibt sich ein ähnliches Bild [Illustration 9]. Hier ist die Load des nachgelagerten HBase Clusters gegenüber der Last des TSD vernachlässigbar. Bei fünf Threads ist durchschnittlich ein Kern zu 70% ausgelastet. Es reichen jedoch Peaks bis hin zu zwei voll ausgelasteten Prozessoren. Diese Maximallast verändert sich auch bei Verwendung von zehn Threads wenig, jedoch ist hier die Grundlast mit 1,0 bereits deutlich höher [Illustration 10].

Auch die Speichernutzung ist wenig überraschend [Illustration 11]. Zum Zeitpunkt des Starts des Benchmarks ist ein Peak auf drei Knoten zu sehen (TSD, HBase Master, 1. Region). Interessant ist die äußerst geringe Speicherlast des TSD. Dieser scheint hauptsächlich CPU-Limitiert.

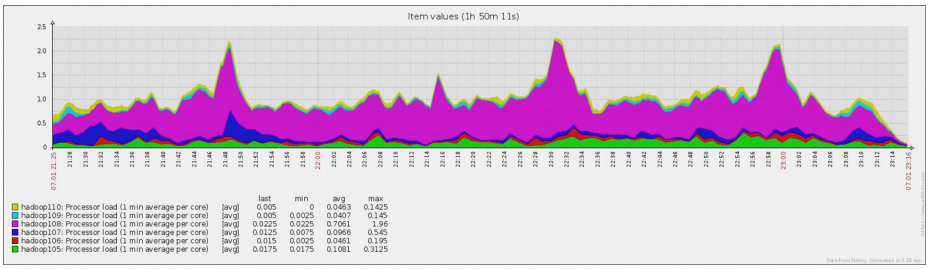

Illustration 9: OpenTSDB 1-Min Load avg; Inserts mit 5 Threads

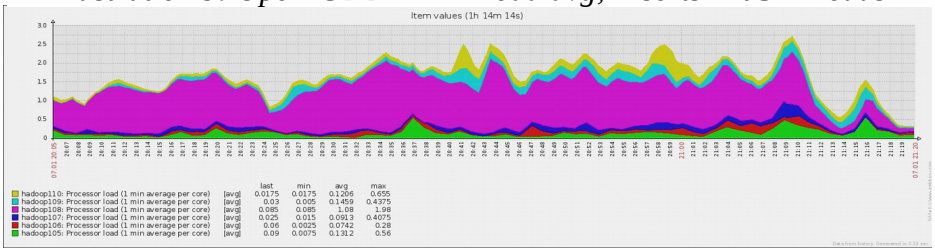

Illustration 10: OpenTSDB 1-Min Load avg; Inserts mit 10 Threads

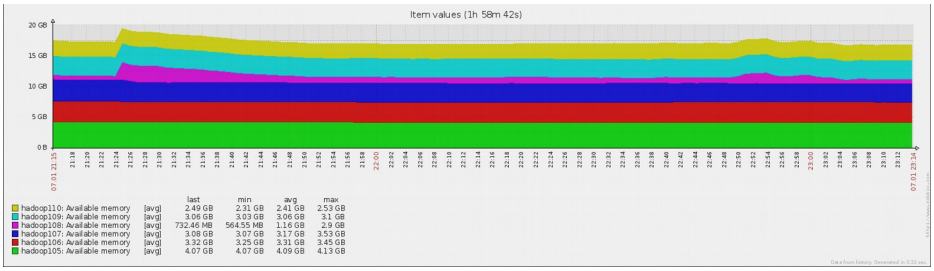

Illustration 11: OpenTSDB freier Speicher; Inserts 5 Threads

HBase

Betrachtet man den Festplattendurchsatz für HBase, so fällt auf, dass sich der Festplattendurchsatz auf allen drei Systemen nach anfänglichen Peaks auf ein sehr niedriges, gleich verteiltes Maß weiter unter 10MB/s einpendelt [Illustrationen 12, 13, 14]. Diese deutet auf eine gleichmäßige Verteilung der Daten hin.

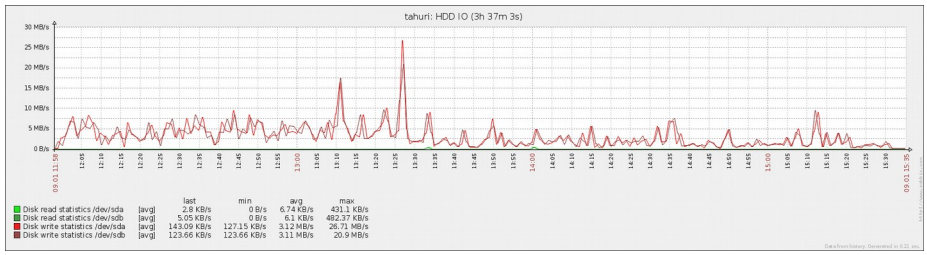

Illustration 12: HBase Festplattendurchsatz Tahuri 5 Threads

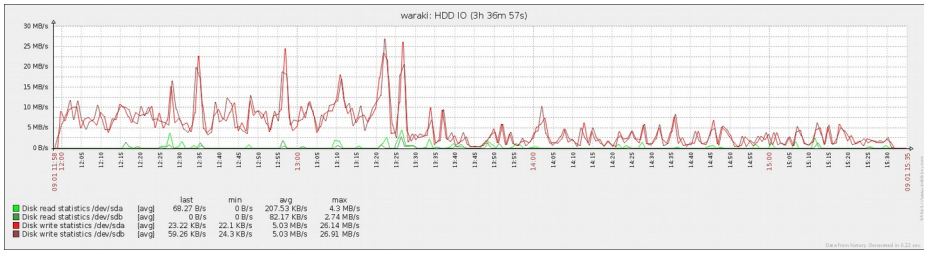

Illustration 13: HBase Festplattendurchsatz Waraki 5 Threads

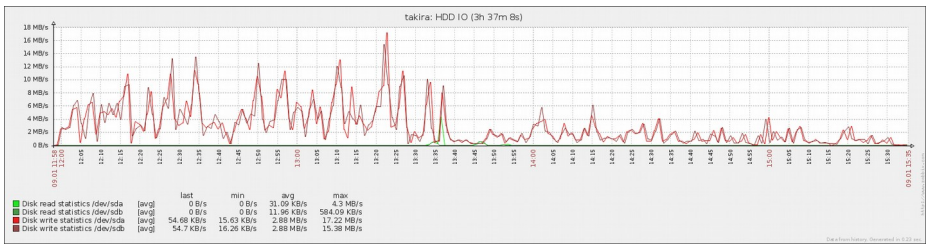

Illustration 14: HBase Festplattendurchsatz Takira 5 Threads

Betrachtet man den Netzwerkdurchsatz [Illustration 15] ergiebt sich ein ähnliches Bild. Nach einer anfänglichen Mehrbelastung auf dem Knoten hadoop110 ist auch hier zu sehen, dass sich die Last, nachdem die Regions verteilt wurden, relativ gleichmäßig verteilt.

Bei der Prozessorlast [Illustration 16] ergibt sich ein ganz ähnliches Bild. Auch hier findet nach einer gewissen Zeit eine gleichmäßige Verteilung nach anfänglichen Peaks auf einer einzelnen Maschine statt. Bei einer maximalen Load von 0,98 ist jedoch auch hier noch reichlich Leistungsreserve vorhanden.

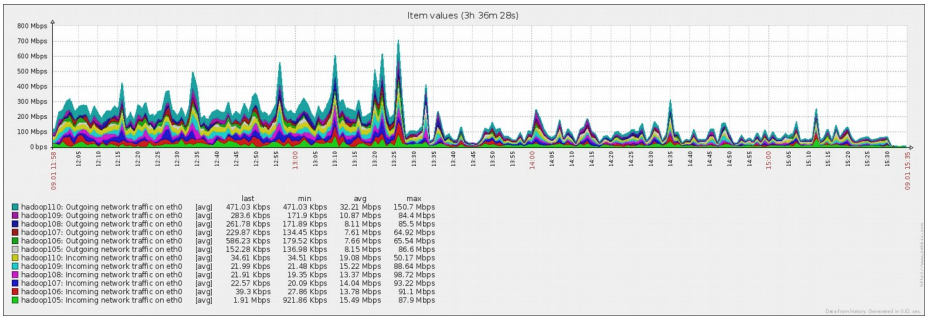

Illustration 15: HBase Netzwerkdurchsatz; Inserts mit 5 Threads

Auch bei der Speichernutzung lässt sich hier kein Flaschenhals erkennen. Im Gegenteil ist die sehr geringe Speicherauslastung etwas überraschend, da HBase versucht möglichst viele Daten im RAM zu cachen um bei Lesevorgängen Festplattenzugriff zu vermeiden. Möglicherweise wurde die initiale Größe dieser Caches nie überschritten.

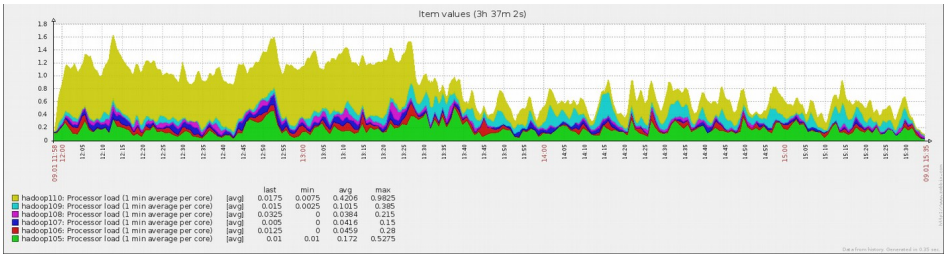

Illustration 16: HBase 1-min Load avg; Inserts mit 5 Threads

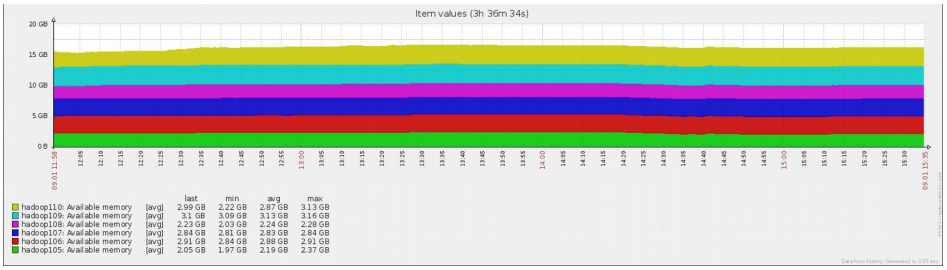

Illustration 17: HBase freier Speicher; Inserts mit 5 Threads

MongoDB

Deutlich zu sehen ist hier die durchweg hohe Schreibrate auf Takira [Illustration 20], wo auch mng1a1, also dem Master des ersten Shards liegt. Die Grundlast allen Knoten kommt durch die Replikation zustande. Deutlich sichtbar durch die starken Peaks sind die Zeitpunkte, zu denen der Region Balancer seine periodische Datenverteilung durchführt. Insbesondere im Verlauf des 05.01. bis zum 06.01. führt dieser eine größere Umverteilung durch [Illustrationen 18, 19, 20].

Generell sieht dieses Verhalten aber nicht optimal aus, was klar wird, wenn man die Graphen betrachtet, die bei zehn Threads entstanden sind [Illustrationen 21, 22, 23]. Dort sind periodisch „Treppen“ zu sehen, die darauf hinweisen, dass der Balancer korrekt arbeitet und regelmäßig Chunks verteilt. Auffällig ist die hohe Schreibrate: Daran lässt sich deutlich erkennen, dass der Schreibaufwand für Daten und Indizes deutlich höher ausfällt als bei HBase, wo keine Indizes gebildet werden.

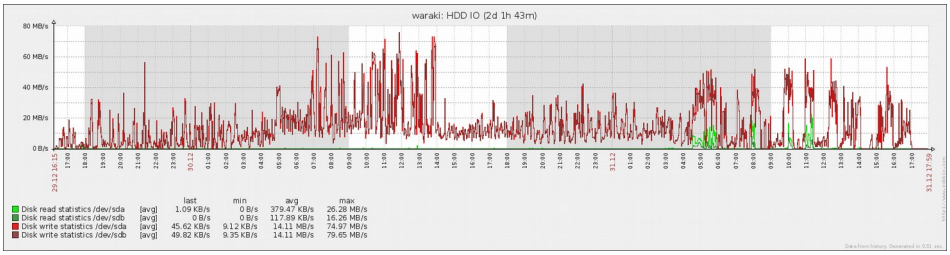

Illustration 18: MongoDB Festplattendurchsatz Waraki 5 Threads

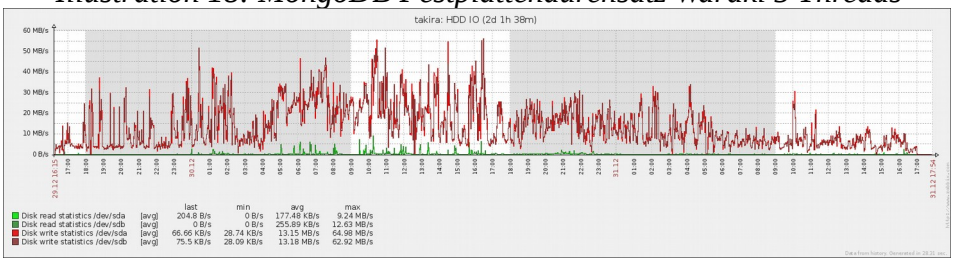

Illustration 19: MongoDB Festplattendurchsatz Takira 5 Threads

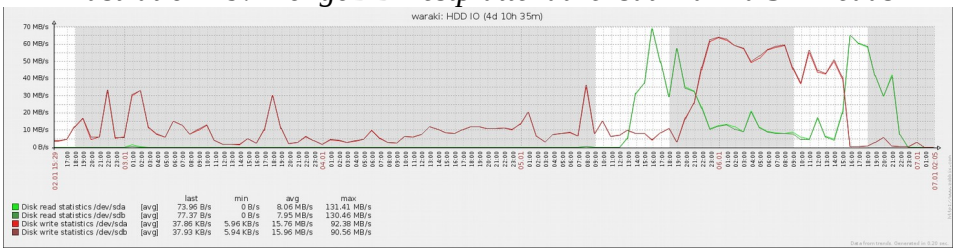

Illustration 20: MongoDB Festplattendurchsatz Waraki 5 Threads

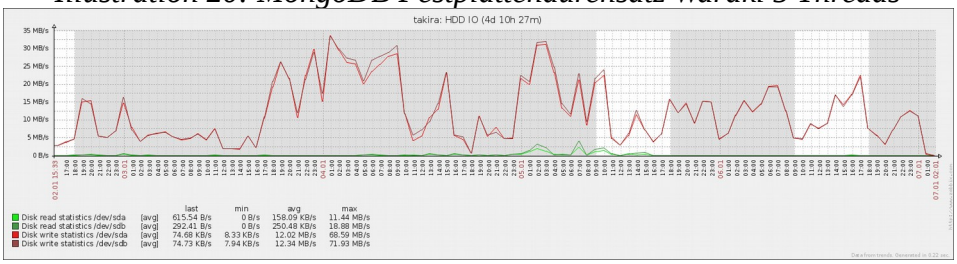

Illustration 21: MongoDB Festplattendurchsatz Takira 10 Threads

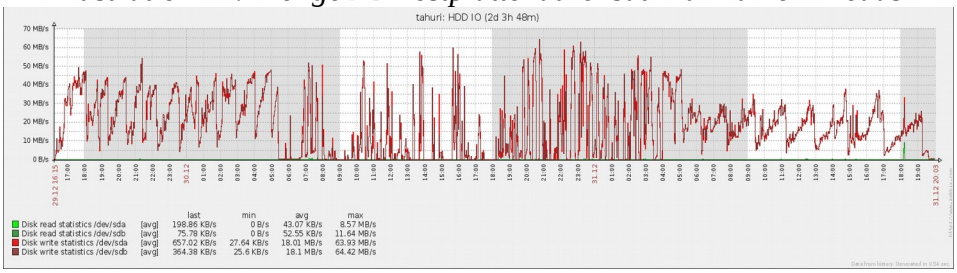

Illustration 22: MongoDB Festplattendurchsatz Tahuri 10 Threads

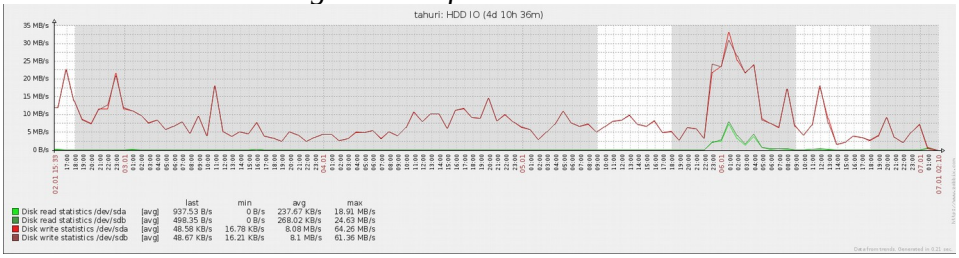

Illustration 23: MongoDB Festplattendurchsatz Tahuri 10 Threads

Die Netzwerklast ist generell sehr niedrig [Illustration 24]. Dies spricht einerseits für eine effiziente Übertragung der Daten hin zur Datenbank. Andererseits ist dies bei der erreichten, sehr niedrigen Insertrate nicht weiter verwunderlich.

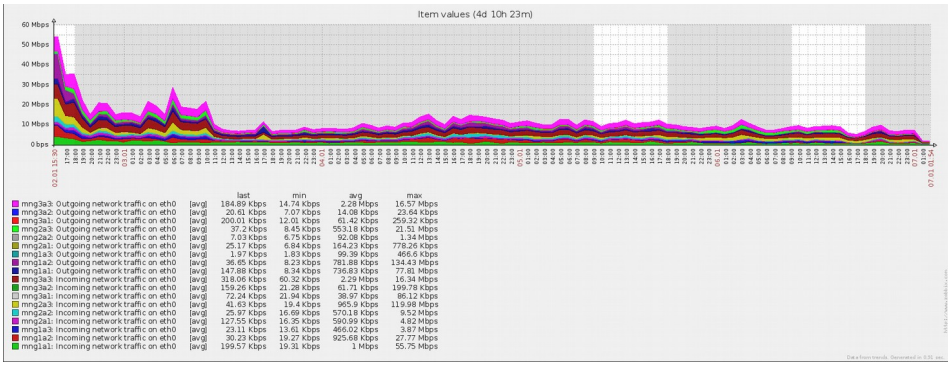

Illustration 24: MongoDB Netzwerkdurchsatz; Inserts mit 5 Threads

Auch bewegt sich die durchschnittliche Load sowohl bei fünf [Illustration 25], wie auch bei zehn schreibenden Threads [Illustration 26] in einem akzeptablen Rahmen. Verglichen mit der Load von HBase oder OpenTSDB ist diese jedoch unter Berücksichtigung der Insertrate sehr hoch.

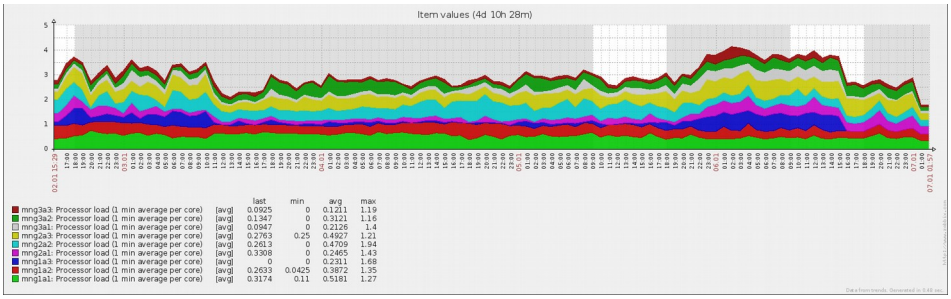

Illustration 25: MongoDB 1-min Load avg; Inserts mit 5 Threads

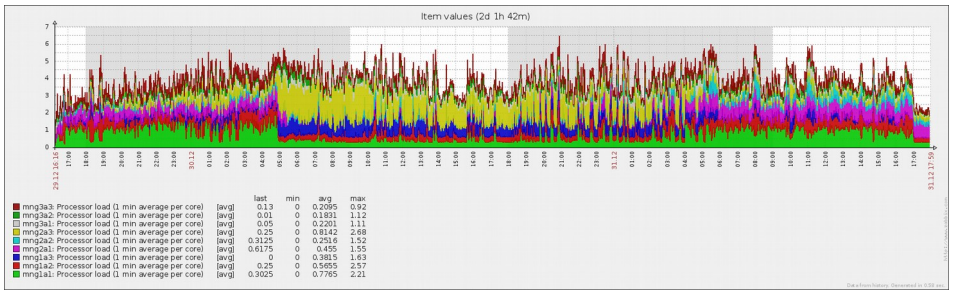

Illustration 26: MongoDB 1-min Load avg; Inserts mit 10 Threads

Die Speicherauslastung ist dagegen sowohl mit fünf [Illustration 27], wie auch mit zehn Threads [Illustration 28] sehr hoch. In den durchgeführten Tests könnte diese durchaus ein Flaschenhals gewesen sein, der eine bessere Skalierung verhindert hat.

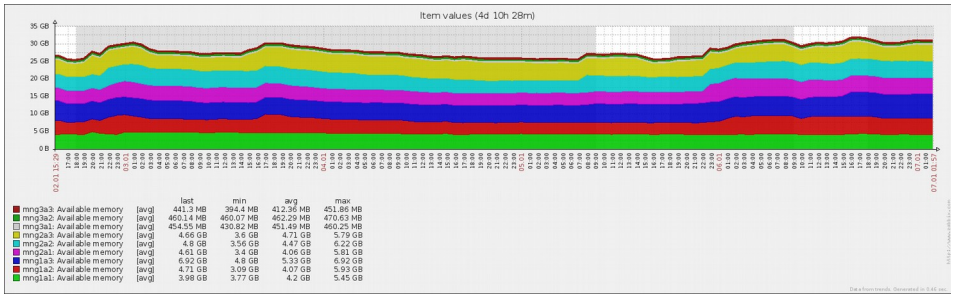

Illustration 27: MongoDB freier Speicher; Inserts mit 5 Threads

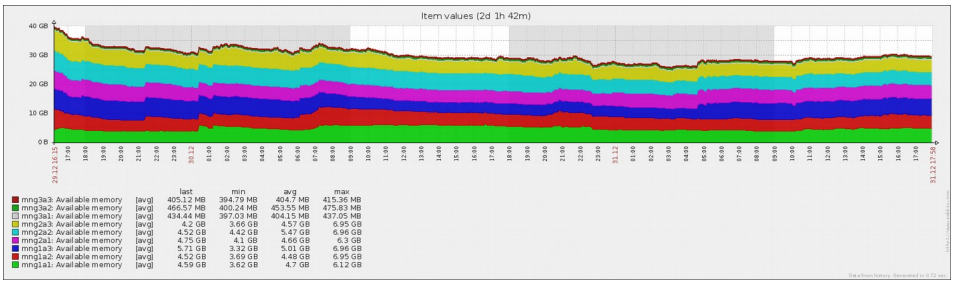

Illustration 28: MongoDB freier Speicher; Inserts mit 10 Threads

5.1.2 Lesen aus der Datenbank

Beim Lesen verschwimmen die starken Unterschiede in der Leistungsfähigkeit etwas [Illustration 29]. Zu beachten ist dabei, dass in der Spalte „NONE“, die keinerlei Aggregation darstellt, für OpenTSDB eine Durchschnittsaggregation gewählt wurde, da OpenTSDB die Ausgabe von Rohdaten nicht unterstützt.

Es fallen an dieser Stelle mehrere Dinge auf. Zum einen bleibt MongoDBs Antwortgeschwindigkeit beim reinen Anfragen ohne Aggregation relativ konstant, egal ob eine Stunde an Daten oder ein ganzes Jahr angefragt werden, während Aggregationen durch die Vergrößerung des Datensatzes langsamer werden.

Zugleich fällt auf, dass es bei HBase und OpenTSDB offenbar kaum eine Rolle spielt, ob aggregiert wird oder nicht. Ebenfalls nicht bemerkbar macht sich der Fakt, dass OpenTSDB die Daten nachträglich aggregiert. Was die beiden Systeme jedoch gemein haben ist, dass die Bearbeitungsdauer der Anfragen mit steigender Datenmenge langsamer wird. Insbesondere fällt dies bei den einfachen Anfragen ohne Aggregation und bei OpenTSDB auf. Dies ist darin begründet, dass HBase Scans, sofern diese nicht von einem Coprocessor ausgelöst werden, seriell ausgeführt werden, um die erwartete Ergebnisreihenfolge garantieren zu können. Die Ursache der auffällig langen HBase

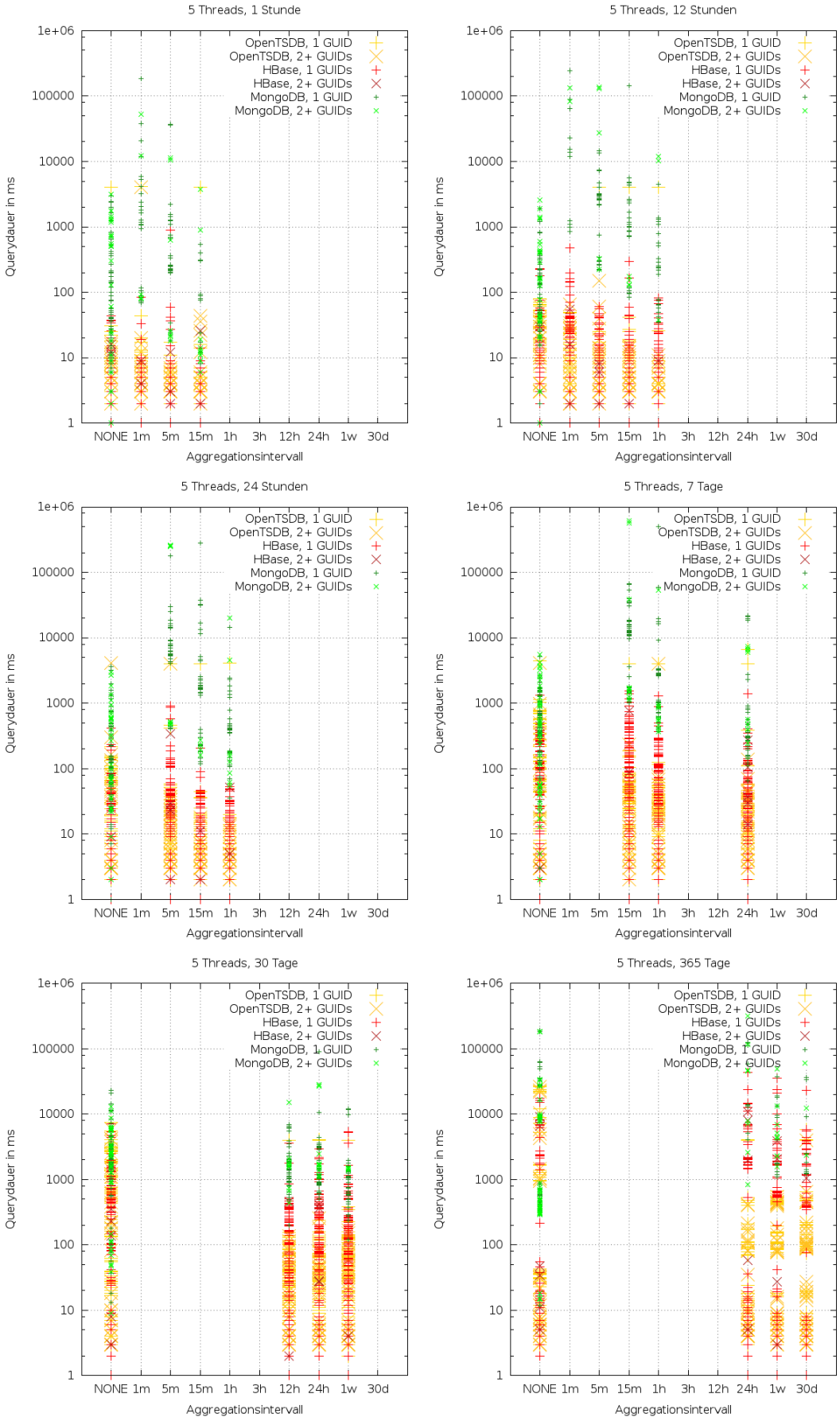

Illustration 29: Lesen verschiedener Queries, 5 Threads

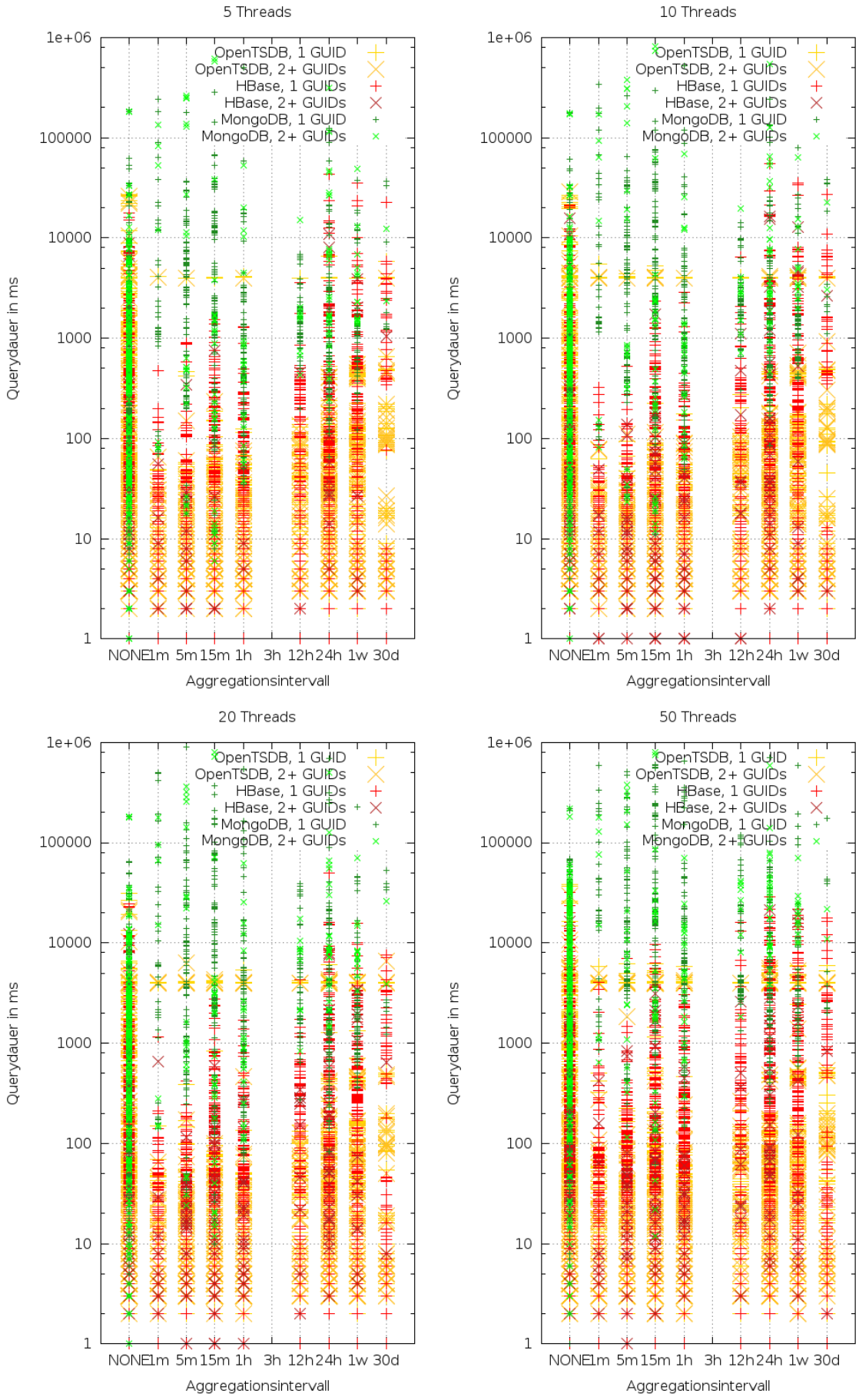

Illustration 30: Gegenüberstellung lesender Queries nach Threadanzahl

Anfragen über den ein-Jahres Zeitraum sind jedoch ohne weiteres nicht nachzuvollziehen.

Betrachtet man die Verteilung der Anfragezeiten im Verhältnis zur Threadanzahl [Illustration 30] fällt auf, dass diese sich kaum ändert. Der Aufwand, um 50 Anfragen parallel zu beantworten scheint sich auf den ersten Blick kaum von dem benötigten Aufwand für fünf Anfragen zu unterscheiden.

OpenTSDB

Tatsächlich trifft dies bei OpenTSDB auch zu. Die erzeugte Last ist bei den Läufen mit fünf Threads [Illustration 31] und denen mit 50 Threads [Illustration 32] kaum zu unterscheiden. Tatsächlich hebt sie sich gerade so von der Grundlast ab.

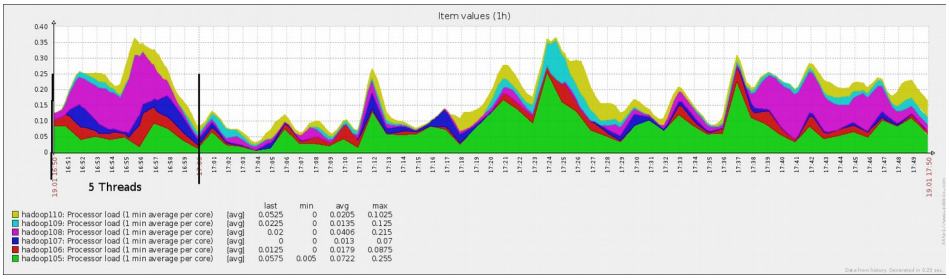

Illustration 31: OpenTSDB 1-min Load avg; 5 Threads Lesend

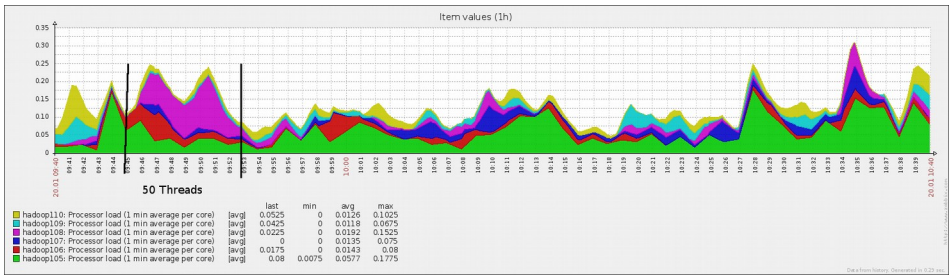

Illustration 32: OpenTSDB 1-min Load avg; 50 Threads Lesend

Die Netzwerklast ist auch hier wieder in einem überschaubaren Rahmen. Von einer gesonderten Betrachtung wird daher abgesehen, da hier kein Flaschenhals zu erwarten ist.

Auch die Speichernutzung erfährt, wie schon bei den Insert-Tests kaum eine Veränderung, weshalb auch hier von einer gesonderten Betrachtung abgesehen wird.
Erwähnenswert ist jedoch das Geschehen auf der Festplatte. Hier kann HBase und damit auch OpenTSDB beim Lesen glänzen. Durch den von HBase verwendeten Bloom-Filter auf Key-Ebene ist kein Lookup auf der Festplatte nötig, um herauszufinden ob Daten

existieren oder nicht. So müssen diese lediglich, falls sie existieren, gelesen werden. Dabei kommt es nicht zu Leseraten jenseits 3MB/sec.

Bei diesem Test trat das in 4.5 bereits genannte Problem auf, dass OpenTSDB Anfragen nicht immer beantwortet. Dies führte dazu, dass Lese-Threads häufig durch einen Timeout beendet werden mussten. Die Gesamtlaufzeit ist dadurch wenig repräsentativ, da dieser Timeout lang genug sein muss um auch Anfragen mit langer Laufzeit nicht zu unterbrechen. Zudem führt es zu potenziell weniger Last, da Threads so teilweise recht lange Idle sind. In Illustration 29 und 30 konnten diese fehlerhaften Anfragen jedoch aus dem Datensatz entfernt werden. Diese sind also um solche falschen Ergebnisse bereinigt.

HBase

Die Last, die von den Anfragen erzeugt wird, steigt bei HBase stetig, hält sich jedoch auch bei 50 Anfragen noch in einem äußerst unbedenklichen Rahmen [Illustrationen 33, 34]. Erst mit 50 Threads entstehen Peaks, die eine Load Average von 1.0 überschreiten.

Wie schon bei OpenTSDB wird von einer gesonderten Betrachtung des Netzwerkdurchsatzes und der Speichernutzung abgesehen. Der Netzwerkdurchsatz erreicht bei 50 Threads mit insgesamt 1,3GBit seinen höchsten Wert, dieses überrascht aber wenig. Das RPC-Protokoll ist für die Übertragung vieler Key-Values wenig platzsparend, da jeder übertragene KeyValue als Tupel aus Key, Column family, Column qualifier, Zeitstempel, Typ, Version und Wert besteht [vgl. HBase JavaDoc, Cell].

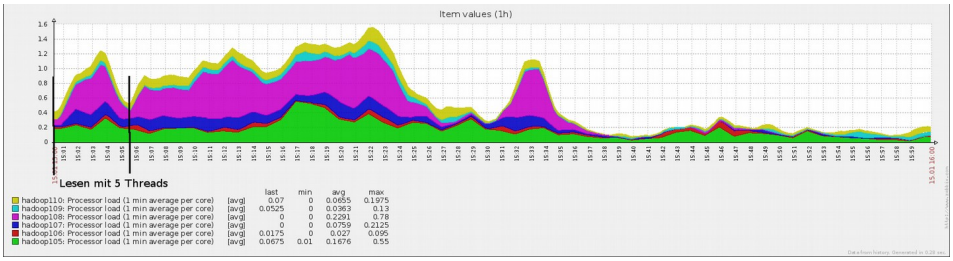

Illustration 33: HBase 1-min Load avg; 5 Threads Lesend

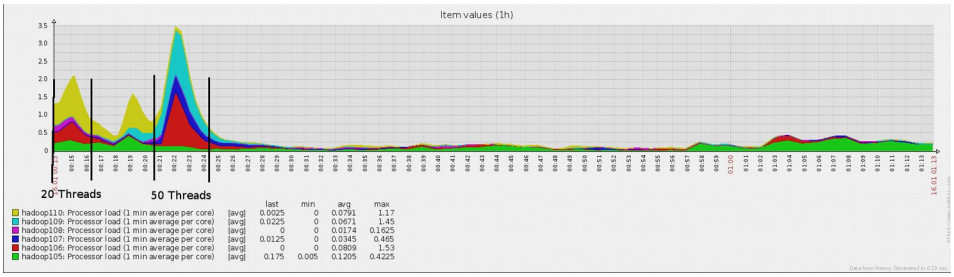

Illustration 34: HBase 1-min Load avg; 20 und 50 Threads Lesend

MongoDB

Die Laststeigerung von fünf auf 50 Threads bei MongoDB ist überraschend [Illustrationen 35, 36]. Während die Last unter Verwendung von fünf Threads in einem normalen Rahmen liegt, steigt diese auf einem Knoten kontinuierlich an, bis dieser bei 50 Threads unter enormem Stress steht. Ursache dafür dürfte SERVER-16700 sein. [vgl. MongoDB Issue Tracker. SERVER-16700] Eine zeitnahe Behebung für einen erneuten Test scheint jedoch ausgeschlossen. Das Problem soll erst in MongoDB 2.9 behoben werden.

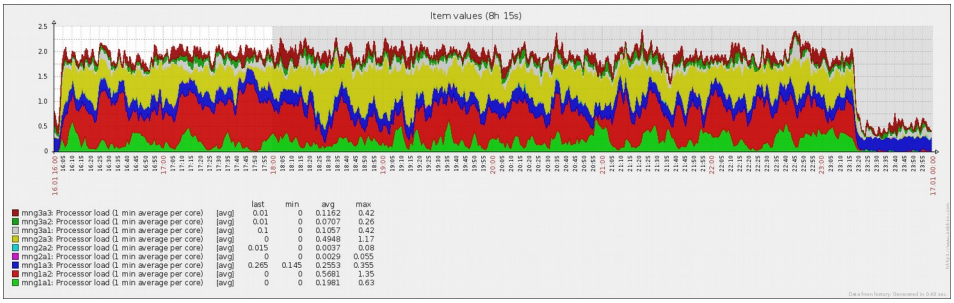

Illustration 35: MongoDB 1-min Load avg; 5 Threads Lesend

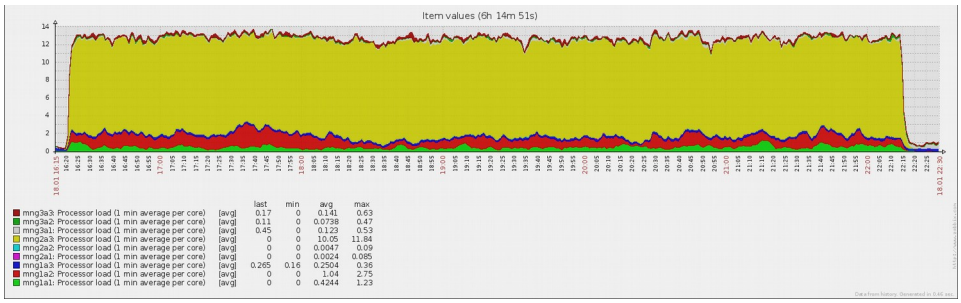

Illustration 36: MongoDB 1-min Load avg; 50 Threads Lesend

Bei der Speichernutzung, dem Netzwerkdurchsatz und dem Festplattendurchsatz ist jedoch auch hier nichts Überraschendes zu sehen. Insbesondere die Leserate auf der Festplatte bleibt über den gesamten Testverlauf Konstant bei ca. 400KB/s.

5.1.3 Gemischte Workload

Hier wird nochmals das Lesen bei einer parallel anliegenden Schreiblast getestet. Diese Tests beschränken sich auf fünf Schreibthreads und 20 bzw. 50 Lesethreads. Dies hat mehrere Gründe:

Zum einen haben die Schreibtests bereits gezeigt, dass zumindest HBase und OpenTSDB bereits mit fünf Threads eine höhere Schreibrate aufweisen als benötigt wird, zum anderen würde mit mehr schreibenden Threads die Last auf den Festplatten zu sehr steigen. Durch die Verwendung eines Raid 1 auf jedem Server wird verteilen sich dort die Zugriffe nicht über mehrere Laufwerke, so kommt es durch diese Tests zu dem Fall, dass bis zu 55 Threads um den Zugriff auf eine Festplatte konkurrieren. Dies mag einem Szenario entsprechen in dem die Datenbank sehr gefordert ist, sollte aber in einem ausreichend dimensionierten System nur sehr selten vorkommen.

Zudem erschienen lesende Tests mit weniger lesenden Threads insbesondere bei OpenTSDB wenig sinnvoll, da durch regelmäßige Fehler in der Beantwortung von Anfragen ein kleiner Thread-Pool häufiger blockiert ist und damit Anfragen nur noch sequentiell oder schlimmstenfalls für bis zu 30 Sekunden gar nicht mehr gestellt werden.

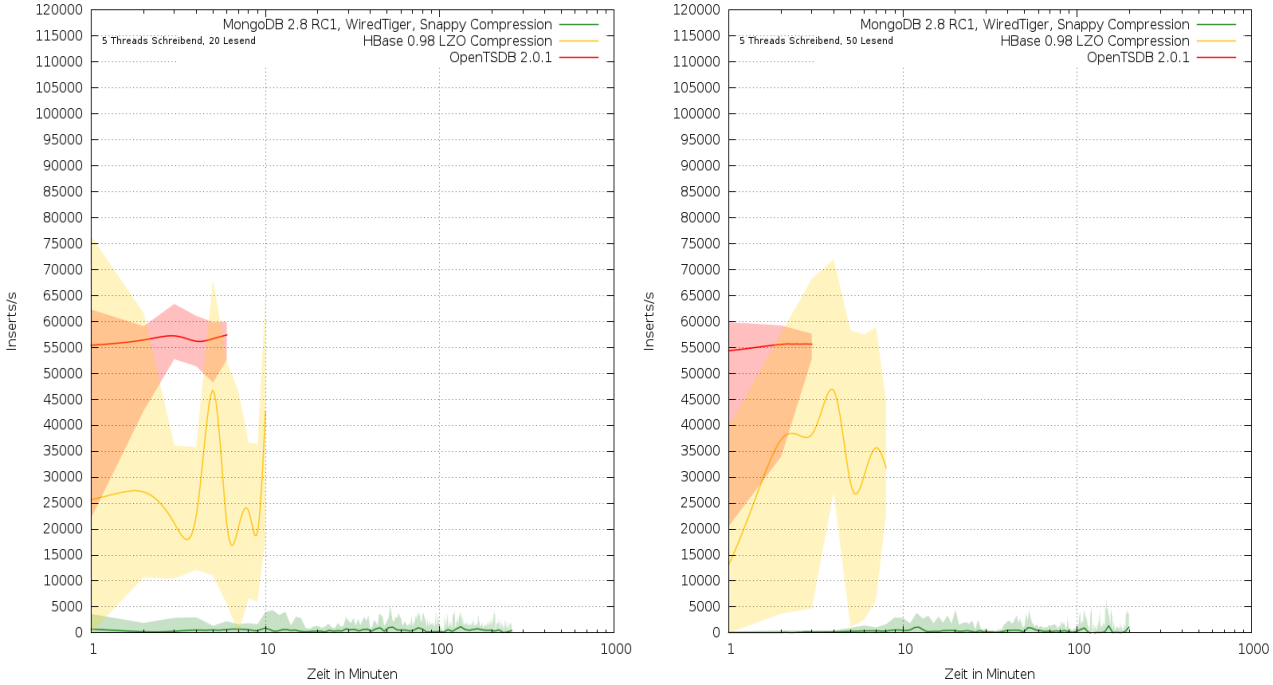

Illustration 37: Inserts Absolut mit 20, bzw. 50 parallelen Lesevorgängen

Gut zu sehen ist hier die weite Streuung der Insertrate von HBase. OpenTSDB kommt in beiden Fällen auf ungefähr die gleiche Insertrate wie bei den reinen Schreibtests.

MongoDB verliert hier nochmals an Leistung gegenüber der Schreibmessung [Illustration 37].

Betrachtet man die Dauer der einzelnen Inserts, so ergibt sich ein ähnliches Bild [Illustration 38]. MongoDBs Latenz steigt deutlich gegenüber den in 5.1 durchgeführten Tests, HBase legt leicht zu.

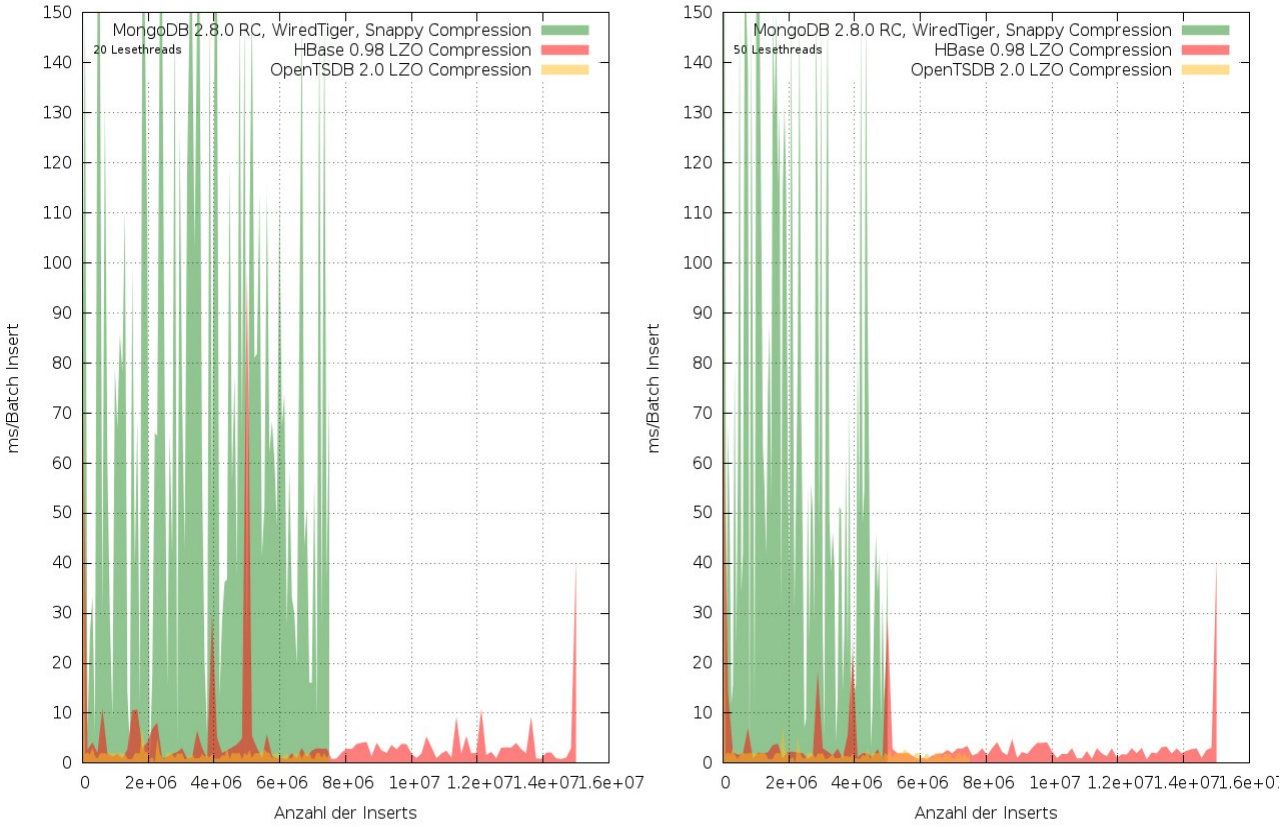

Illustration 38: Durchschnittliche Insertdauer mit 20, bzw. 50 parallelen Lesevorgängen

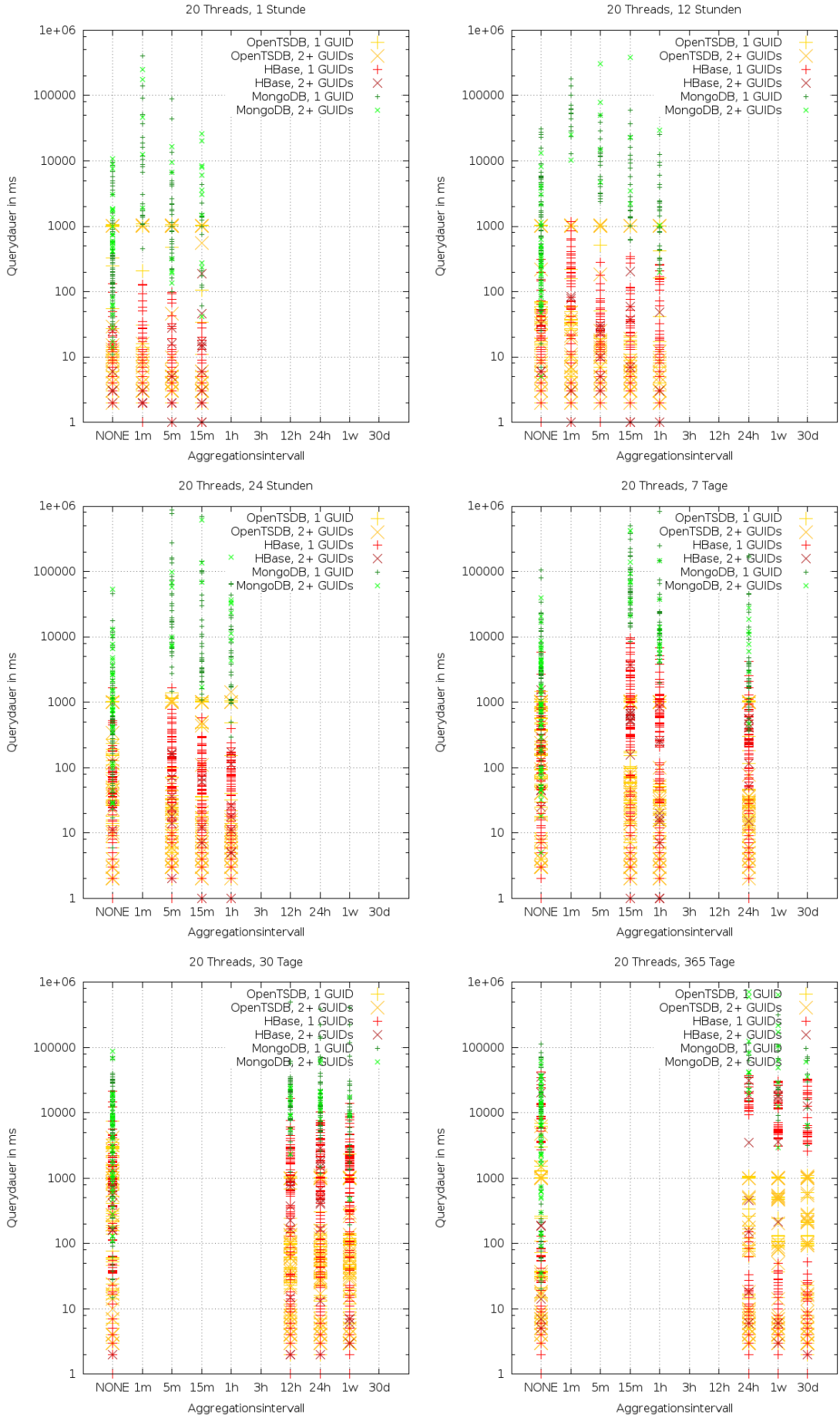

Illustration 39: Lesen verschiedener Queries, 20 Threads

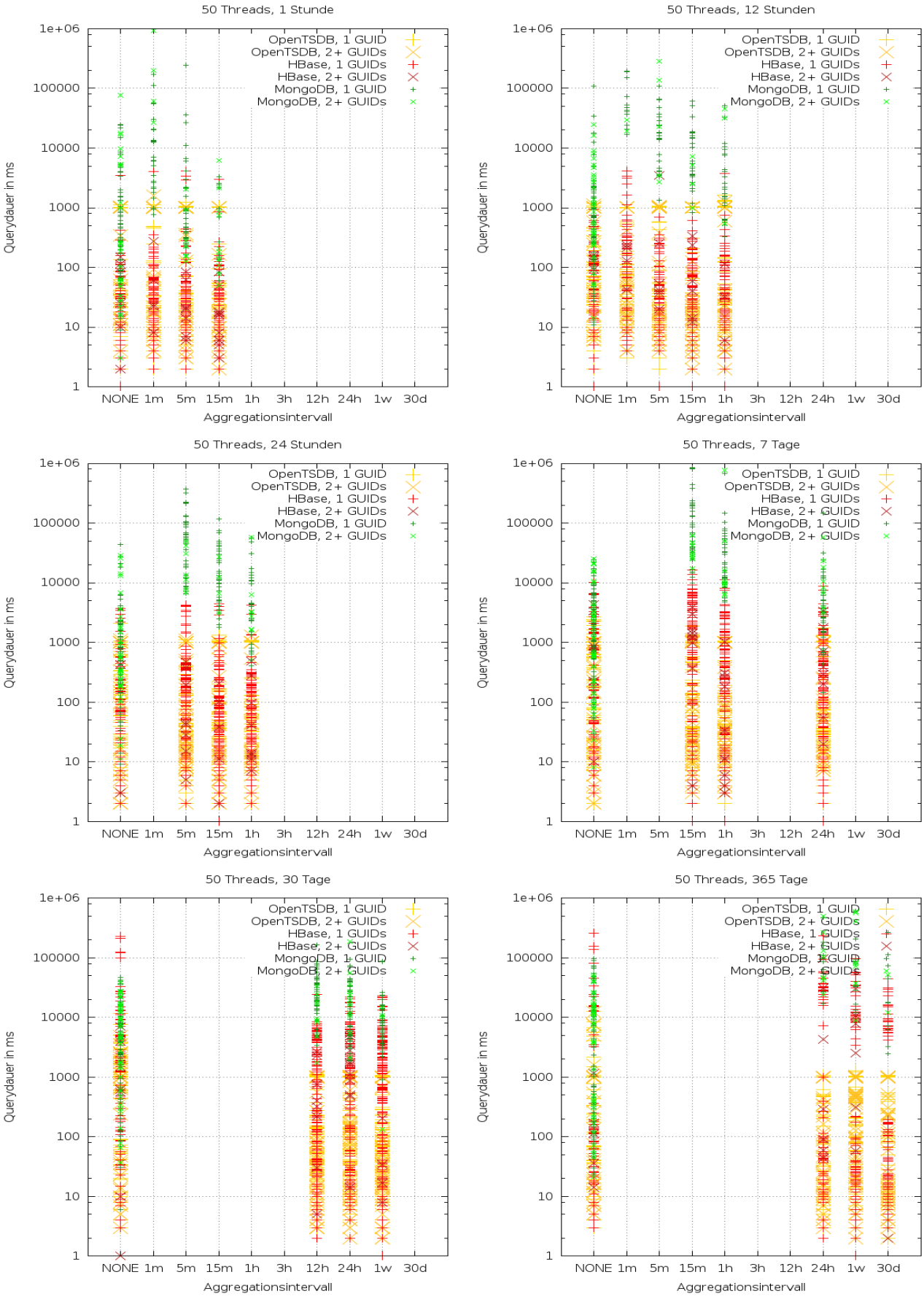

Illustration 40: Lesen verschiedener Queries, 50 Threads

Größere Unterschiede offenbaren sich dagegen bei der Leserate. Bei 20 Threads steigt diese bei HBase und OpenTSDB mit der Größe des Datenbereichs stetig an, MongoDB dagegen zeigt wieder eine über den gesamten Test konstante Antwortzeit [Illustration 39]. Während MpngpDB gegenüber den vorausgegangenen Tests kaum verliert, ist der Trend bei OpenTSDB und HBase deutlicher.

Betrachtet man die Querydauer unter Verwendung von 50 Threads [Illustration 40], ist ein nochmaliger Anstieg der Dauer zu beobachten. Durch die sehr gemischte Last aus Lesen- und Schreiben mit einer großen Threadanzahl und die Tatsache, dass in diesen Tests maßgeblich von der Festplatte gelesen wird, kommt es andauernd zu Disk-Seek. Diese Operation dauert laut Seagate 8.5, bzw. 9.5ms (Lesend/Schreibend) [Seagate. Datenblatt Constellation ES], eine Messung mit Hilfe des Programms seeker, ergab jedoch Werte zwischen 12,9 und 14,2ms. Es muss in allen Systemen außerdem gerade bei umfangreicheren Anfragen aufgrund der Datenverteilung im Cluster potenziell von mehreren Quellen gelesen werden, wodruch sich Disk-Seeks weiter summieren. So kommt es mit steigender Threadzahl zu einer kontinuierlichen Verlangsamung. Dem kann durch die Verwendung mehrerer Festplatten und das Aufteilen der Daten auf diese entgegengewirkt werden.

Von einer nochmaligen Betrachtung der Monitorngdaten kann für diese Tests abgesehen werden. Wenig überraschend bewegen diese sich im gleichen Rahmen der vorangegangenen Tests. So zeigt sich auch bei MongoDB in der Load wieder der Einfluss von SERVER-16700 [vgl. MongoDB Issue Tracker. SERVER-16700]. Potenzielle Bottlenecks sind aus diesen ansonsten nicht erkennbar. Der vermutete Engpass durch Disk-Seeks kann aus diesen auch nicht ermittelt werden. Um dies zu messen, müsste man eher die Dauer bestimmter Funktionsaufrufe in den Datenbanken messen, was den Rahmen dieser Auswertung sprengt.

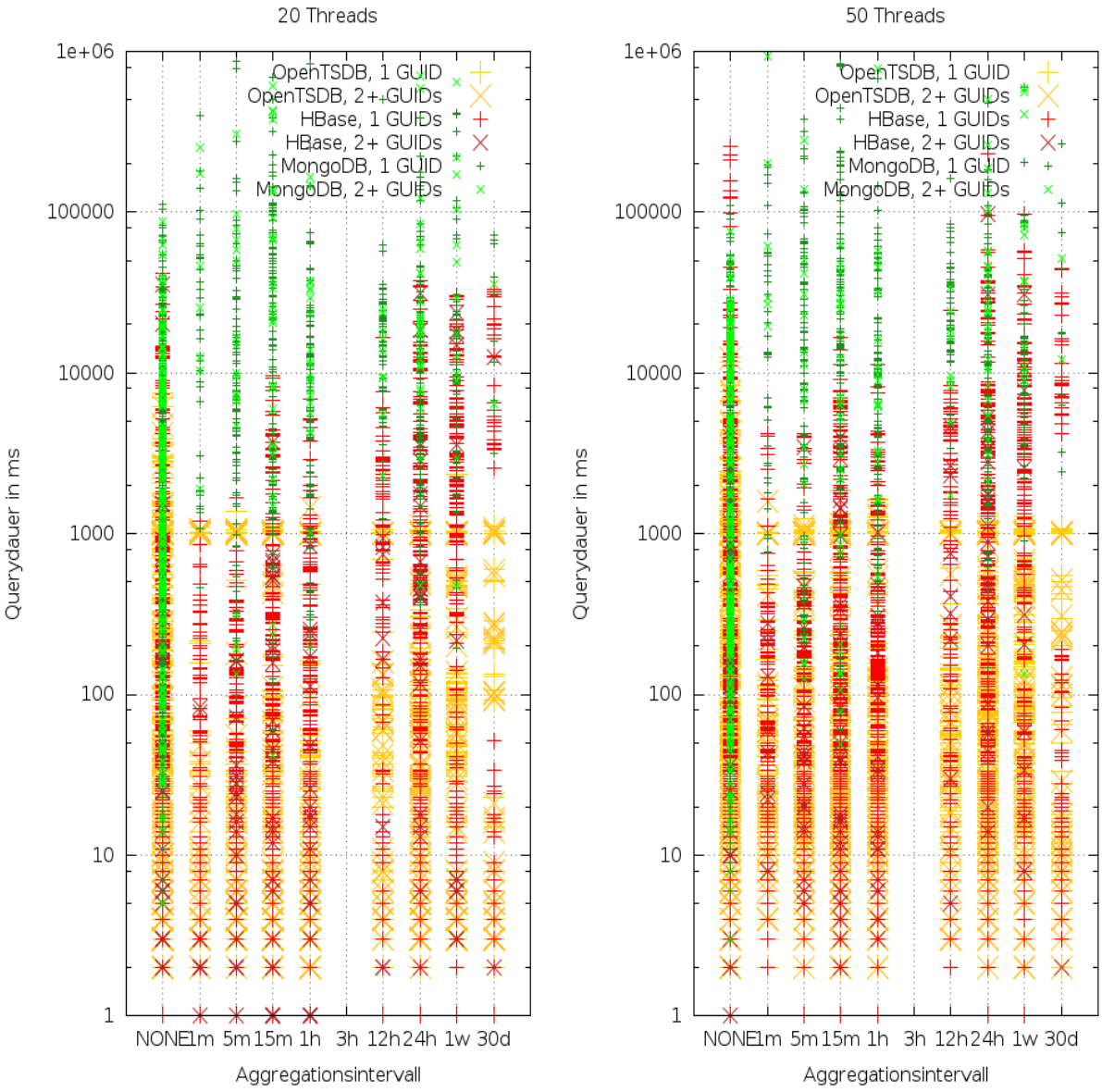

Illustration 41: Gegenüberstellung lesender Queries nach Threadanzahl

5.2 Sonstige Beobachtungen

5.2.1 Platzbedarf

Die von den Systemen importierten 350.911.200 Datenpunkte haben ohne Berücksichtigung von Redundanz folgenden Speicherbedarf:

	HBase	**OpenTSDB**	**MongoDB**
Unkomprimiert	14,749 GB	8,661 GB	-
Komprimiert	2,804 GB	2,048 GB	10,715 GB

5.2.2 Simulierter Knotenausfall

5.2.2.1 HBase

Ausfall eines Masters

Es kommt zum Failover auf einen anderen Master.

Ausfall eines RegionServers

Anfragen, die gegen diesen RegionServer gestellt werden, gehen verloren. Es erfolgt eine Recovery durch einen anderen RegionServer. Dadurch kommt es aufgrund der eventuell nicht mehr gegebenen Datenlokalität zu einer temporären Verlangsamung. Datenverlust kann dadurch verhindert werden, dass das Write-Ahead Log in HDFS von allen Knoten verwendet wird.

5.2.2.2 MongoDB

Ausfall eines Replication Masters

Die Chance von Datenverlust ist in diesem Fall hoch. Daten, die auf den Master geschrieben wurden, aber noch nicht repliziert wurden, gehen verloren. Es findet ein Failover auf einen anderen Knoten statt, neue Schreibvorgänge werden von diesem bearbeitet.

Ausfall eines Slaves

Es kommt eventuell zu Verlangsamungen beim Lesen, dadurch, dass mehr Last auf die verbliebenen Server übertragen werden muss.

5.3 Erkenntnisse aus der Evaluation

Im direkten Vergleich zeigt sich, dass OpenTSDB und dessen asynchroner Client in Sachen Insertgeschwindigkeit bei weitem am schnellsten sind. Daraus ist auch ersichtlich, dass HBase noch wesentlich weiter skalieren kann, als dies mit dem klassischen HTable-Client hier ersichtlich wird. Jedoch schlägt auch dieser sich sehr gut. Es ist anzunehmen, dass die Maximalleistung beider Clients hier noch lange nicht erreicht wird und der Sweet-Spot von HTable, also der Punkt, an dem der Thread-Pool am effektivsten arbeitet, noch jenseits der in diesem Test gefahrenen zehn Threads liegt. MongoDB skalierte in den hier durchgeführten Tests beim Schreiben sogar nahezu linear über die Zeit, konnte jedoch nicht annähernd die Leistungsfähigkeit von HBase oder OpenTSDB erreichen.

Beim Lesen verschwimmen dann die großen Unterschiede. Hier schneidet MongoDB bei der Abfrage roher Daten akzeptabel ab, verliert aber bei der Aggregation deutlich gegenüber HBase und OpenTSDB. In diesem Bereich sind alle Systeme den gestellten Anforderungen durchaus gewachsen, wenn es auch bei OpenTSDB nicht beantwortete Anfragen gibt.

Insgesamt lässt sich aber sagen, dass nur HBase den gestellten Anforderungen gewachsen ist. HBase hat in den durchgeführten Tests bewiesen, dass es sowohl die gestellten Leistungsanforderungen, wie auch die gestellten Stabilitäts- und Qualitätsanforderungen erfüllen kann, auch wenn dies durch einen erhöhten Implementierungsaufwand erkauft wird. Denn Unterstützung für bestimmte Datentypen muss erst geschaffen werden, auch jegliche Form komplexerer Aggregationen muss selbst implementiert werden, wie das Beispiel TAggregator zeigt. Darüber hinaus muss auch gesagt werden, dass die durchgeführten Tests zwar gezeigt haben, dass, während MongoDB bereits mit dem verwendeten Datensatz Skalierungsprobleme hatte, dieser

noch zu klein war um in Bereiche vorzudringen, wo HBase optimal skaliert, geschweige denn Skalierungsprobleme auftreten.

So ist OpenTSDB sicher die richtige Wahl für die Infrastrukturüberwachung, wofür es geschaffen wurde. Dort ist irrelevant, dass beim Downsampling der Zeitstempel gemittelt wird und dass die Interpolation in Randbereichen nicht ganz korrekt ist, oder dass Anfragen manchmal zu einem Timeout führen. Für ein Portal, das täglich von zahlreichen Kunden genutzt wird, sind dies jedoch Ausschlusskriterien.

Auch wenn MongoDB grundsätzliche Vorteile wie eine Query Language bietet und Indexierung erlaubt, was in manchen Nutzungsszenarien auch wichtig ist, so überwiegen doch die Probleme. Es gibt deutliche Defizite in Sachen Stabilität und Leistungsfähigkeit in einem verteilten Setup. Auch reicht die Skalierarkeit lange nicht an HBase heran. Die Gründe dafür sind sicherlich in der Implementierung selbst zu finden. MongoDB ist sehr ähnlich einer klassischen, relationalen Datenbank. Die damit einhergehende Komplexität übersteigt HBase, welches lediglich ein Key-Value Store ist, um Längen.

6. Verteilte Zeitseriendatenbank – Quo vadis?

6.1 Zusammenfassung

Im Rahmen dieser Arbeit sollte ermittelt werden, welches der getesteten Systeme sich am besten für den Einsatz als Data-Warehouse System für Zeitseriendaten eignet. Die Wahl fiel auf diese Systeme, weil sie eine große Entwicklergemeinde haben, in der sie sich einen sehr guten Ruf erarbeitet haben und quelloffen sind.

Zunächst wurden dazu Testszenarien erarbeitet, die auf den aktuellen Anforderungen an das bisher verwendete Warehouse-System aufbauen.
Als Hilfsmittel, diese zu untersuchen, wurden Programme zum Lesen- und Schreiben aus allen getesteten Systemen entwickelt. Darüber hinaus entstand der TAggregator, ein einfacher HBase-Coprocessor für dieAggregation von Zeitseriendaten.

Nachdem diese Arbeiten abgeschlossen waren, wurden die Systeme anhand der erarbeiteten Testszenarien getestet und miteinander verglichen. Dabei kristallisierte sich HBase als klarer Favorit heraus. Mit MongoDB konnte trotz einer Migration von Version 2.6 auf 2.8-RC aufgrund verschiedener Bugs kein Testsetup aufgebaut werden, welches das volle Potenzial des Systems zeigt. Gleichzeitig sind diese auch der Grund dafür, dass das System nicht empfohlen werden kann.

6.2 Handlungsempfehlung

Empfohlen wird die Verwendung von HBase als Warehouse-Lösung. Zudem sollte evaluiert werden, inwieweit bestehende und zukünftige Systeme sich in den Hadoop-Stack integrieren lassen, um so eine einheitliche Plattform zu schaffen.

Für den Betrieb wird weiterhin empfohlen Hadoop und HBase nicht in virtuellen Maschinen Maschinen zu betreiben. Die Anforderungen an Festplattendurchsatz und vorhandenen Arbeitsspeicher rechtfertigen die Installation auf dedizierter Hardware. So kann Last auf mehrere Festplatten verteilt werden und vorhandener Arbeitsspeicher für das Caching von Daten genutzt werden. Die Systeme sollten sollten zwischen vier und acht Festplatten als

JBOD zur Verfügung stellen, die über nicht weniger als 16GB Ram verfügen und idealerweise mit Gbit Ethernet an der gleichen Switch untereinander verbunden sind, um Latenzen möglichst klein zu halten.

6.3 Ausblick

Auf Basis dieser Arbeit soll nun das zukünftige Warehouse-System der Kiwigrid GmbH entstehen. Dazu zeigt die Arbeit mit HBase das geeignetste System und einen guten Ansatz für die Weiterentwicklung des Datenbankschemas auf. Denkbar ist eine iterative Entwicklung einzelner Komponenten, wie der Insert-Logik, der Aggregationslogik und weiterer Komponenten bis hin zum fertigen Produkt.

Essentiell werden dabei zum einen die Integration in das bestehende System und die Integration neuer Anforderungen, wie Disaggregation, die parallel zu dieser Arbeit entstanden sind. Dabei bietet das System jedoch auch an, Lösungsansätze wie MapReduce zu integrieren, die im Rahmen dieser Arbeit nicht behandelt werden konnten.

Auch bietet sich die Möglichkeit, bestehende Komponenten in den Hadoop-Stack zu integrieren und so für die Zukunft eine einheitliche Cloud-Lösung aufzubauen (2.8).

Inwieweit sich Hadoop als Plattform für eine Vielzahl von Aufgaben durchsetzen wird, kann jedoch nur die Zeit zeigen.

Glossar

B/B+ Bäume:

B+ Bäume speichern Daten in Pages. Eine Page wird beim erreichen einer bestimmten Größe geteilt und die resultierenden Pages miteinander verknüpft. Das erlaubt sehr effizientes Einfügen und Suchen von Daten. Das Problem ist jedoch, dass diese Pages auf der Festplatte nicht notwendigerweise nahe zusammen liegen. Dies führt bei größeren Datenbanken zu einer Limitierung der Performance durch die notwendigen Seeks der Festplatte. [vgl. George, L. 2011, S. 315f]

LSM Bäume:

LSM (Log Structured Merge) Bäume speichern Daten sequentiell in ein Log. Von dort werden sie in eine In-Memory Datenstruktur übertragen. Sobald genug Updates auf einen Bereich dieser Struktur angesammelt wurden, werden diese sortiert in einem neuen Storefile auf der Festplatte abgelegt. So entsteht eine Datenstruktur ähnlich einem B-Baum, die jedoch auf sequentielles Lesen- und Schreiben optimiert ist. [vgl. George, L. 2011, S. 316f]

Load Average

Die Load Average ist ein Maß für die Prozessorlast. Dabei bedeutet 0.0, dass keine Last vorhanden ist. Ein Wert von 1.0 steht für einen voll ausgelasteten Prozessorkern, 2.0 für zwei ausgelastete Kerne, usw. Diese Metrik wird vom Linux Kernel geführt und errechnet sich aus der Anzahl der Prozesse, die darauf warten, von der CPU ausgeführt zu werden. Sie ist nicht als isolierter Wert zu einem bestimmten Zeitpunkt, sondern als ein Trend über einen bestimmten Zeitraum zu verstehen. Die 1-min load average beschreibt demnach die Durchschnittliche Last über eine Minute. [vgl. Walker, R. 2006]

Quellenangaben

Apache Hadoop. 2013. Apache Hadoop NextGen MapReduce. Verfügbar unter: http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html [14.12.2011]

Apache HBase. 2015. Home. Verfügbar unter: http://hbase.apache.org/ [26.01.2015]

Lai, M., Koontz, E., Purtell, A. 2012. HBase Coprocessors Introduction. Verfügbar unter: https://blogs.apache.org/hbase/entry/coprocessor_introduction [12.12.2014]

Apache HBase Dokumentation. 2015. Verfügbar unter: http://hbase.apache.org/book.html [23.01.2015]

Apache Issue Tracker.

HADOOP-6311. Verfügbar unter: https://issues.apache.org/jira/browse/HADOOP-6311 [02.12.2014]

HBASE-5157. Verfügbar unter: https://issues.apache.org/jira/browse/HBASE-5175 [01.12.2014]

Chang, F., et al. 2008. Bigtable: A Distributed Storage System for Structured Data. ACM Transactions on Computer Systems. Volume 26 Issue 2, June 2008 Article No. 4

Codd, E.F. 1970. A relational model of data for large shared data banks. Communications of the ACM. Volume 13 Issue 6. S. 377-387

Comerford, A., 2014. MongoDB 2.8: New WiredTiger Storage Engine adds compression. Verfügbar unter: http://comerford.cc/wordpress/2014/11/12/mongodb-2-8-new-wiredtiger-storage-engine-adds-compression/ [10.01.2015]

Dean, J., Ghemawat, S. 2004. MapReduce: Simplified Data Processing on Large Clusters. OSDI'04: Sixth Symposium on Operating System Design and Implementation. Verfügbar unter: http://research.google.com/archive/mapreduce.html [15.01.2015]

Dimiduk, N., Khurana A. 2013. HBase in Action. Shelter Island: Manning

George, L. 2011. HBase the Definitive Guide (First Edition). Sebastopol: O'Reilly Media

Ghemawat, S., Gobioff, H., Leung, S. 2003. The Google File System. ACM SIGOPS Operating Systems Review - SOSP '03. S. 29-43

Gilbert, S., Lynch, N. 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. ACM SIGACT News.Volume 33, Issue 2. S 51-59

Gogate V. 2014. Hadoop configuration & performance tuning. Verfügbar unter: http://de.slideshare.net/vgogate/hadoop-configuration-performance-tuning [02.12.2014]

Hadoop Wiki. Too Many Open Files. Verfügbar unter: https://wiki.apache.org/hadoop/TooManyOpenFiles [02.12.2014]

Hawkins, D. 2010. JVM Internals. Verfügbar unter: http://de.slideshare.net/dougqh/jvm-internals-key-note [26.01.2015]

HBase 0.98 APIDocs. AggregationClient. Verfügbar unter: http://archive.cloudera.com /cdh5/cdh/5/hbase/apidocs/org/apache/hadoop/hbase/client/coprocessor/AggregationClient.html [26.01.2015]

Hetzner Wiki, EX-Server, verfügbar unter: http://wiki.hetzner.de/index.php/Root_Server_Hardware#EX-Server [25.01.2015]

Kamat, G., Singh, S. 2013. Compression Options in Hadoop - A Tale of Tradeoffs. Verfügbar unter: http://de.slideshare.net/Hadoop_Summit/kamat-singh-june27425pmroom210cv2 [20.01.2015]

MongoDB Blog. 2014. Announcing MongoDB 2.8.0-RC0 Release Candidate and Bug Hunt. Verfügbar unter: http://blog.mongodb.org/post/102461818738/announcing-mongodb-2- 8-0-rc0-release-candidate-and [25.01.2015]

MongoDB Blog. On Distributed Consistency. Verfügbar unter: http://blog.mongodb.org/post/475279604/on-distributed-consistency-part-1 [12.01.2015]

Schema Design for Time Series Data in MongoDB. Verfügbar unter: http://blog.mongodb.org/post/65517193370/schema-design-for-time-series-data-in-mongodb [08.12.2014]

MongoDB Documentation.

Hadoop Use Cases. Verfügbar unter: http://docs.mongodb.org/ecosystem/use-cases/hadoop/ [13.01.2015]

Mongos. Verfügbar unter: http://docs.mongodb.org/manual/reference/program/mongos/#bin.mongos [12.12.2014]

Record Allocation Strategies. Verfügbar unter: http://docs.mongodb.org/manual/core/storage/#record-allocation-strategies [26.01.2015]

Replica Set Arbiter. Verfügbar unter: http://docs.mongodb.org/manual/core/replica-set-

arbiter/ [12.12.2014]

Sharded Collection Balancing. Verfügbar unter: http://docs.mongodb.org/manual/core/sharding-balancing/#sharding-balancing [12.12.2014]

ulimit. Verfügbar unter: http://docs.mongodb.org/manual/reference/ulimit/ [12.12.2014]

MongoDB Issue Tracker.

SERVER-15691. Verfügbar unter: https://jira.mongodb.org/browse/SERVER-15691 [25.01.2015]

SERVER-16700. Verfügbar unter: https://jira.mongodb.org/browse/SERVER-15691 [21.01.2015]

NIST. 2013. Engineering Statistics Handbook. Verfügbar unter: www.itl.nist.gov/div898/handbook/index.htm [15.01.2015]

OpenTSDB Documentation. Interpolation. Verfügbar unter: http://opentsdb.net/docs/build/html/user_guide/query/aggregators.html#interpolation [14.12.2014]

Redmond, E., Wilson, J. 2012. Seven Databases in Seven Weeks. San Francisco: Pragmatic Programmers

Seagate. Datenblatt Constellation ES. Verfügbar unter: http://www.seagate.com/www-content/product-content/constellation-fam/constellation-es/constellation-es-2/de/docs/constellation-es2-fips-data-sheet-ds1725-5-1207de.pdf [26.01.2015]

Seethana, S. 2014. Docker & Kubernetes on Apache Hadoop YARN. Verfügbar unter: http://hortonworks.com/blog/docker-kubernetes-apache-hadoop-yarn/ [25.11.2014]

Sigoure, B. 2011. OpenTSDB – The Distributed, Scalable, Time Series Database For your modern monitoring needs. Verfügbar unter: http://opentsdb.net/misc/opentsdb-oscon.pdf [04.12.2014]

Sigoure, B. 2012. Lessons Learned from OpenTSDB. Verfügbar unter: http://de.slideshare.net/cloudera/4-opentsdb-hbasecon [15.01.2015]

Sigoure, B. 2014. OpenTSDB 2.0. Verfügbar unter: http://tsunanet.net/~tsuna/hbasecon-2014-opentsdb-2.0.pdf [27.01.2015]

Soboroff, I. 2013. Benchmarking LZO Compression in HBase. Verfügbar unter: http://isoboroff.github.io/posts/benchmarking-lzo-compression-in-hbase.html [10.01.2015]

Tanenbaum, A., van Steen, M. 2007. Distributed Systems: Principles and Paradigms (Second Edition). Upper Saddle River: Pearson Prentice Hall

Universität Würzburg. 2012. A First Course on Time Series Analysis – Examples with SAS (Version 2012.August.01). Institut für Mathematik, Universität Würzburg: Verfügbar unter: http://statistik.mathematik.uni-wuerzburg.de/timeseries/ [20.01.2015]

Walker, R. 2006. Examining Load Average. Verfügbar unter: http://www.linuxjournal.com/article/9001 [26.01.2015]

White, T. 2011. Hadoop: The Definitive Guide (2nd Edition). Sebastopol: O'Reilly Media

Wolpe, T. 2014. MongoDB CTO: How our new WiredTiger storage engine will earn its stripes. Verfügbar unter: http://www.zdnet.com/article/mongodb-cto-how-our-new-wiredtiger-storage-engine-will-earn-its-stripes/ [22.11.2014]

Xavier, M. et. al. 2013. Performance Evaluation of Container-based Virtualization for High Performance Computing Environments. 21St Euromicro International Conference on Parallel, Distributed and Network-Based Processing. S. 233-240

Appendix I

Knoten	Installierte Anwendungen
hdp0.int.kiwidev.com (Server 1)	• HDFS DataNode • Ganglia Monitor • Ganglia Server • HBase Master • HBase RegionServer • HDFS Client • MapReduce2 Client • Nagios Server • NameNode • Pig • Tez Client • YARN Client • ZKFailoverController • ZooKeeper Client • ZooKeeper Server
hdp1.int.kiwidev.com (Server 1)	• App Timeline Server • HDFS DataNode • Ganglia Monitor • HBase RegionServer • HDFS Client • History Server • MapReduce2 Client • Pig • ResourceManager • Tez Client

	• YARN Client • ZooKeeper Client • ZooKeeper Server
hdp2.int.kiwidev.com (Server 2)	• HDFS DataNode • Ganglia Monitor • HBase Client • HBase RegionServer • HDFS Client • JournalNode • MapReduce2 Client • Pig • Tez Client • YARN Client • ZooKeeper Client • ZooKeeper Server
hdp3.int.kiwidev.com (Server 2)	• HDFS DataNode • Ganglia Monitor • HBase Client • HBase RegionServer • HDFS Client • JournalNode • MapReduce2 Client • NodeManager • Pig • Tez Client • YARN Client • ZooKeeper Client

hdp4.int.kiwidev.com (Server 3)	• HDFS DataNode • Ganglia Monitor • HBase Client • HBase RegionServer • HDFS Client • JournalNode • MapReduce2 Client • NodeManager • Pig • Tez Client • YARN Client • ZooKeeper Client
hdp5.int.kiwidev.com (Server 3)	• NodeManager • HDFS DataNode • Ganglia Monitor • HBase Master • HBase RegionServer • HDFS Client • HDFS NameNode • ZKFailoverController

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Facharbeit selbstständig und ohne fremde Hilfe angefertigt und nur die im Literaturverzeichnis angeführten Quellen und Hilfsmittel benutzt habe.

Insbesondere versichere ich, dass ich alle wörtlichen und sinngemäßen übernahmen aus anderen Werken (inkl. Internetinhalte) als solche kenntlich gemacht habe.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Ort, Datum: Unterschrift: